

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

JOÃO LUIS DA SILVA GUIO SOARES

UM ESTUDO SOBRE OS MECANISMOS DE CONCORRÊNCIA DA LINGUAGEM
GO

RIO DE JANEIRO
2019

JOÃO LUIS DA SILVA GUIO SOARES

UM ESTUDO SOBRE OS MECANISMOS DE CONCORRÊNCIA DA LINGUAGEM
GO

Trabalho de conclusão de curso de graduação
apresentado ao Departamento de Ciência da
Computação da Universidade Federal do Rio
de Janeiro como parte dos requisitos para ob-
tenção do grau de Bacharel em Ciência da
Computação.

Orientador: Profa. Silvana Rossetto

RIO DE JANEIRO

2019

CIP - Catalogação na Publicação

S676e Soares, João Luis da Silva Guio
Um estudo sobre os mecanismos de concorrência da
linguagem Go / João Luis da Silva Guio Soares. --
Rio de Janeiro, 2019.
86 f.

Orientadora: Silvana Rossetto.
Trabalho de conclusão de curso (graduação) -
Universidade Federal do Rio de Janeiro, Instituto
de Matemática, Bacharel em Ciência da Computação,
2019.

1. Linguagem de programação (Computadores). 2. Go
(Linguagem de programação de computador). 3. Golang.
4. Concorrência. I. Rossetto, Silvana, orient. II.
Título.

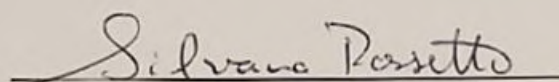
JOÃO LUIS DA SILVA GUIO SOARES

UM ESTUDO SOBRE OS MECANISMOS DE CONCORRÊNCIA DA LINGUAGEM
GO

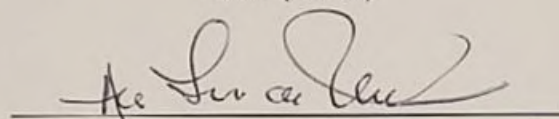
Trabalho de conclusão de curso de graduação
apresentado ao Departamento de Ciência da
Computação da Universidade Federal do Rio
de Janeiro como parte dos requisitos para ob-
tenção do grau de Bacharel em Ciência da
Computação.

Aprovado em 28 de agosto de 2019

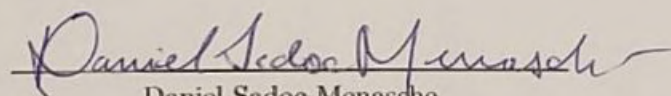
BANCA EXAMINADORA:



Silvana Rossetto
D.Sc. (UFRJ)



Ana Lúcia de Moura
D.Sc. (PUC-Rio)



Daniel Sadoc Menasche
D.Sc. (UFRJ)

Em homenagem a todos que me engradeceram para a conclusão dessa jornada.

AGRADECIMENTOS

Agradeço à todo o corpo docente do departamento de Ciência da Computação da UFRJ, em especial à professora Silvana Rossetto.

RESUMO

Este estudo explora a linguagem Go e seus mecanismos para programação concorrente. Apresenta-se uma visão geral da linguagem, destacando os recursos de concorrência oferecidos por ela. Soluções de problemas clássicos de concorrência e de padrões de computação distribuída são propostos e analisados, mostrando o uso dos diversos recursos providos por Go. As soluções apresentadas demonstram a simplificação de código por meio das abstrações oferecidas pela linguagem e pelo modelo de concorrência baseado em trocas de mensagens via canais de comunicação. Avalia-se também a possibilidade e dificuldades de estender a linguagem para oferecer uma implementação generalizada de corrotinas, usando como referência uma implementação na linguagem Lua. Por fim, apresentamos uma análise simplificada do desempenho do Go, comparando os tempos de execução de um algoritmo de multiplicação de matrizes entre duas soluções: uma desenvolvida em Go, e outra em C++. A linguagem C++ obteve desempenho superior a Go.

Palavras-chave: Concorrência. Linguagens de Programação. Go. Golang.

ABSTRACT

This study explores the Go programming language and its concurrency artifacts. First, it shows basic concepts of concurrency and the mechanisms to create concurrent programs. After that, the study shows the basic usage of the Go language, and dives into the simple yet complete concurrency mechanisms. Then, classic problems solutions and distributed systems classic patterns are implemented in Go are discussed and analyzed, showing the usage of the language's toolkit. All solutions are using the native language toolkit, and it's message passing in shared memory model, using communication channels. The study also analyzes on how to extend the Go language with a coroutine implementation based on Lua coroutines. Finally, a simplified comparison between C++ and Go is made, comparing performance of execution speed in the matrix multiplication problem. C++ outperforms Go in the given problem.

Keywords: concurrency. programming language. go. golang.

LISTA DE ILUSTRAÇÕES

Figura 1 – Representação de Processos e <i>Threads</i>	15
Figura 2 – Condição de <i>deadlock</i> entre fluxos de execução	19
Figura 3 – Escalonamento do sistema operacional	27
Figura 4 – Escalonamento das <i>goroutines</i> dentro de <i>threads</i> do sistema operacional	27
Figura 5 – Fila de <i>goroutines</i> com duas <i>kernel threads</i>	27
Figura 6 – Chegada de uma <i>goroutine</i>	28
Figura 7 – Alocação da <i>goroutine</i> 1	28
Figura 8 – Chegada de uma nova <i>goroutine</i>	28
Figura 9 – <i>Job stealing</i> : <i>goroutine</i> é alocada numa <i>kernel thread</i> diferente da fila a qual pertencia	29
Figura 10 – Novas <i>goroutines</i> criadas	29
Figura 11 – Fluxograma das <i>threads</i> do barbeiro dorminhoco	49
Figura 12 – Tempos de execução da solução em Go com otimização de acesso ao cache	70
Figura 13 – Tempos de execução da solução em C++ com otimização de acesso ao cache	71
Figura 14 – Comparação das soluções em Go e C++, ambas com otimização de cache	72
Figura 15 – Tempos de execução da solução em Go com otimização de cache e com o algoritmo de particionamento da matriz	76
Figura 16 – Comparação das soluções em Go e C++, ambas com otimização de cache e com o mesmo algoritmo	77

LISTA DE CÓDIGOS

2.1	Pseudocódigo de seção crítica	17
3.1	Código “olá mundo” em Go	20
3.2	Exemplo de função anônima concorrente	25
3.3	Exemplo simples de um canal	30
3.4	Comunicando por canais	31
3.5	Exemplo de uso do select	33
3.6	Exemplo de select com leitura não bloqueante	34
4.1	Código do produtor com apenas um consumidor e um produtor	36
4.2	Código do consumidor com apenas um consumidor e um produtor	36
4.3	Código da função principal com apenas um consumidor e um produtor	37
4.4	Código da função principal com N consumidores e M produtores	38
4.5	Código do escritor para implementação básica	39
4.6	Código do leitor para implementação básica	40
4.7	Código da função principal para implementação básica	41
4.8	Código do escritor com prioridade para escrita	43
4.9	Código do leitor com prioridade para escrita	44
4.10	Código da função principal com prioridade para escrita	45
4.11	Código do escritor para implementação sem <i>starvation</i>	47
4.12	Código do leitor para implementação sem <i>starvation</i>	47
4.13	Código da função principal para implementação sem <i>starvation</i>	48
4.14	Código do barbeiro	49
4.15	Código do cliente	50
4.16	Código da função principal do barbeiro dorminhoco	51
4.17	Exemplo de servidor	52
4.18	Exemplo de cliente	53
4.19	Código da função principal do servidor do chat	55
4.20	Código da função de recebimento de novos usuarios do chat . .	57
5.1	Código de definição e criação de corrotinas em Go	60
5.2	Continuação do código de corrotinas: <i>resume</i> e <i>yield</i> das cor- rotinas	61
5.3	Exemplo de corrotinas: percorrendo árvores binárias	63
6.1	Código da estrutura Matrix em Go	66
6.2	Código da função principal de multiplicação de matrizes em Go	67
6.3	Código da função de multiplicação de matrizes em paralelo em C++	67

6.4	Código da função principal de multiplicação de matrizes em C++	68
6.5	Código da estrutura Matrix em Go com o mesmo algoritmo . .	74
6.6	Código da função principal em Go	75
A.1	Exemplo de tipos booleanos	82
A.2	Exemplo de tipos numéricos	83
A.3	Exemplo de texto	83
A.4	Exemplo de array e slice	83
A.5	Exemplo de estrutura	84
A.6	Exemplo de interface e método	84
A.7	Exemplos de mapas	85
A.8	Diferentes usos das estruturas condicionais clássicas	85
A.9	Exemplo do uso de for	85
A.10	Iterando em um array por meio da palavra reservada range . .	86
A.11	Exemplo de interface para comunicação remota	86

SUMÁRIO

1	INTRODUÇÃO	12
2	COMPUTAÇÃO CONCORRENTE	14
2.1	PROCESSOS E THREADS	14
2.1.1	<i>Kernel thread vs User-space thread</i>	15
2.2	SINCRONIZAÇÃO E COMUNICAÇÃO ENTRE FLUXOS DE EXECUÇÃO IN-DEPENDENTES	15
2.2.1	Condição de corrida	16
2.2.2	Sincronização por condição lógica	18
2.2.3	<i>Deadlocks & Starvations</i>	18
3	A LINGUAGEM DE PROGRAMAÇÃO GO	20
3.1	ESPECIFICAÇÕES	20
3.1.1	Packages	20
3.1.2	Tipos de dados	21
3.1.3	Controles de fluxo	23
3.1.4	Interfaces e Métodos	24
3.2	RECURSOS PARA CONCORRÊNCIA	25
3.2.1	Goroutines	25
3.2.1.1	Escalonamento de <i>goroutines</i>	26
3.2.2	Channels	29
3.2.3	A biblioteca <i>sync</i>	30
3.2.4	Select	32
4	USO DOS RECURSOS DE CONCORRÊNCIA DE GO	35
4.1	PROBLEMAS CLÁSSICOS DE CONCORRÊNCIA	35
4.1.1	Produtor e consumidor	35
4.1.1.1	Um produtor e um consumidor com <i>buffer</i> de tamanho mínimo	35
4.1.1.2	N produtores e M consumidores, com <i>buffer</i> de tamanho qualquer	37
4.1.2	Leitores e escritores	38
4.1.2.1	Implementação básica	39
4.1.2.2	Leitores e escritores com prioridade para escrita	41
4.1.2.3	Leitores e escritores sem starvation	45
4.1.3	Barbeiro dorminhoco	48
4.2	PROGRAMAÇÃO DISTRIBUÍDA	51
4.2.1	Aplicação cliente/servidor	52

4.2.2	Chat	54
5	CORROTINAS	59
5.1	IMPLEMENTAÇÃO EM GO	60
5.2	EXEMPLO DE USO DE CORROTINAS EM GO	62
6	AVALIAÇÃO DE DESEMPENHO DE GO	65
6.1	MULTIPLICAÇÃO DE MATRIZES	65
6.2	CÓDIGO EM GO	65
6.3	COMPARAÇÃO COM OUTRA LINGUAGEM	67
6.4	OTIMIZAÇÃO DE ACESSO AO CACHE	69
6.5	MEDIÇÕES	69
6.5.1	Desempenho por núcleo	69
6.5.2	Considerações na compilação	72
6.6	COMPARAÇÃO COM O MESMO ALGORITMO	73
7	CONCLUSÃO	78
	REFERÊNCIAS	81
	APÊNDICE A – CÓDIGOS DE EXEMPLOS DA LINGUAGEM GO	82

1 INTRODUÇÃO

Computação concorrente, ou computação de fluxos concorrentes, é um campo de estudo que surgiu a partir da necessidade de permitir execuções de vários programas simultaneamente em uma mesma máquina. Os sistemas operacionais são um dos exemplos de uso mais clássicos desse tipo de computação: o computador moderno inicializa executando um único programa, que dará início a um sistema operacional capaz de permitir que diversos outros programas executem simultaneamente, concorrendo pelos recursos da máquina, como processador e espaço de memória.

Nos últimos anos, o uso de máquinas com múltiplos *cores* de processamento tem se tornado cada vez maior. A partir da década de 2010, encontra-se disponível no mercado *notebooks*, *tablets*, *smartphones*, computadores pessoais, entre outros dispositivos, que disponibilizam algum tipo de processador que possui mais de um núcleo de processamento. Simultaneamente, o aumento do uso e da complexidade de certas aplicações (como por exemplo aplicações Web) aumentaram as necessidades das empresas por desenvolver sistemas de melhor desempenho, que utilizem ao máximo os recursos computacionais da máquina hospedeira.

Nesses novos cenários, a computação concorrente torna-se cada vez mais relevante. Ao mesmo tempo que a tecnologia de hardware evoluiu, novas ferramentas e abstrações de linguagem de programação concorrente foram disponibilizadas. Muitas linguagens, porém, tiveram que fazer diversas adaptações para prover um melhor ferramental de computação concorrente, pois não estava presente em seu projeto inicial. A linguagem Python, por exemplo, passou a oferecer os recursos de *async/await*, que permite a escrita de códigos assíncronos, com o lançamento da versão 3.5, em setembro de 2015 (PRANSKEVICHUS; SELIVANOV, 2015). A linguagem Java criou um modelo de *green threads*, um tipo de mecanismo de concorrência que facilita o uso de fluxos concorrentes, sem utilizar múltiplos processadores, e só depois disponibilizou *threads* nativas do Java (CORPORATION, 2000). C++ disponibilizou uma biblioteca própria de *threads* apenas na versão de 2011 (ISO, 2011), antes tendo de utilizar *threads* POSIX, ou a biblioteca *pthreads* do C.

A linguagem Go, por sua vez, já foi projetada pensando em concorrência: primitivas da própria linguagem fornecem mecanismos para criar e escalonar novos fluxos de execução, e diferentes alternativas para comunicação de forma segura entre os fluxos de execução concorrentes. Como descrito na página inicial da linguagem, Go propõe-se em ser uma linguagem que facilita a construção de programas simples, confiáveis e eficientes (PIKE et al., 2019b).

O objetivo deste trabalho é estudar os mecanismos de concorrência oferecidos pela linguagem Go, explorando o seu uso em diferentes padrões de programação concorrente. Esperamos avaliar as facilidades providas por esses mecanismos para o desenvolvimento de

problemas clássicos de concorrência, assim como a possibilidade de usá-los para estender a linguagem provendo uma implementação generalizada do conceito de corrotinas, usando como referência uma implementação na linguagem Lua.

O modelo de troca de mensagem oferecido por Go, utilizando a estrutura de dados chamada de canal, é um mecanismo poderoso de sincronização de fluxos, com ordenação e atomicidade garantida pela linguagem. Como será demonstrado, implementações do padrão produtores/consumidores tornam-se tão simples quando uma leitura ou escrita de uma variável (nesse caso, um envio ou recebimento dentro de um canal). O modelo de comunicação via canais proposto pela linguagem, porém, não resolve completamente a necessidade de acesso simultâneo a um determinado recurso: para cenários onde modelar a solução do problema por meio de troca de mensagens não é óbvia, Go fornece ferramentas clássicas de sincronização.

O restante deste trabalho está organizado da seguinte forma. Inicialmente, no capítulo 2, são apresentados os conceitos básicos de computação concorrente e seus principais desafios. No capítulo 3 apresentamos uma visão geral da linguagem Go, incluindo suas características mais básicas e suas abstrações para programação concorrente. Continuando, no capítulo 4 avaliamos as abstrações de concorrência e o desempenho de Go implementando alguns problemas clássicos de programação concorrente, paralela e distribuída. Avaliamos também uma proposta de extensão da linguagem, introduzindo o mecanismo de corrotinas, não oferecido por padrão. Por fim, o capítulo 7 apresenta as conclusões do trabalho.

2 COMPUTAÇÃO CONCORRENTE

Concorrente, na sua etimologia, vem do latim *concurrents*, que significa "que corre junto" (CUNHA, 2019). Em computação, está comumente associado a ideia de não dependência de ordem de execução entre os programas, em que fluxos diferentes não tem uma dependência causal entre eles, podendo assim ser fluxos concorrentes (LAMPORT, 1978).

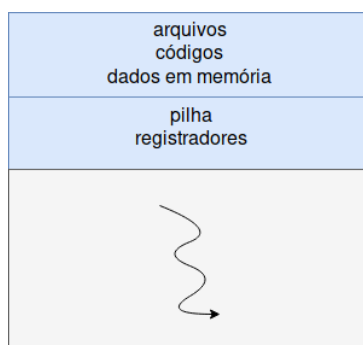
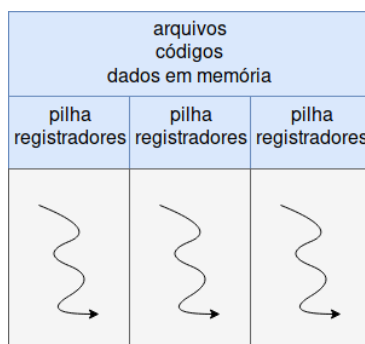
2.1 PROCESSOS E THREADS

Processos são a abstração oferecida pelo sistema operacional para um programa em execução. O sistema operacional fica responsável pela execução concorrente dos diversos processos em uma mesma máquina, permitindo o uso convencional de computadores hoje, em que um ou mais usuários executam diversos programas ao mesmo tempo (BRYANT; RICHARD; RICHARD, 2003).

É por meio da abstração de processos que um programa se comunica com as outras entidades, podendo ser *hardwares* disponíveis ou mesmo outros processos. Em sistemas operacionais modernos, cada processo recebe um espaço de memória para sua execução, evitando problemas de integridade dos dados do seu programa. As comunicações com outros dispositivos são muitas vezes feitas por chamadas de sistemas (*syscall*), onde um programa criado por um usuário requisita ao sistema operacional algum tipo de ação, como por exemplo, desenhar na tela uma imagem.

Os processos, porém, não são a menor abstração de um fluxo de execução de um programa em sistemas operacionais modernos. Um processo possui uma ou mais *threads*, ou fluxos de execução independentes, sendo o espaço de memória e o código de um processo compartilhada entre elas, enquanto cada uma possui seus próprios registradores e ponteiros de instrução. Esta disposição de recursos está demonstrada na figura 1. Todo processo possui, então, pelo menos uma *thread*, comumente chamada de *main thread*, ou fluxo de execução principal. Toda *thread* utiliza o espaço de memória do processo. Assim, *threads* compartilham memória entre si, em contrapartida dos processos, que não possuem memória compartilhada. Uma *thread* não existe fora de um processo.

Threads podem ser coordenadas de maneira preemptiva, quando há uma disputa pelo uso de recurso (núcleos de processamento), ou cooperativamente, em que a troca de fluxo é controlada diretamente pelas próprias *threads*. As *threads* cooperativas possuem este nome devido suas características de cessar seu próprio fluxo de execução, cedendo espaço para um outro fluxo. Em sistemas operacionais modernos, as *threads* do sistema operacional são comumente preemptivas. Elas são também chamadas de *kernel threads*, e são mais detalhadas na seção 2.1.1.

Figura 1 – Representação de Processos e *Threads*(a) Uma única *thread* por processo(b) Várias *threads* por processo

2.1.1 *Kernel thread vs User-space thread*

Existem dois tipos principais de *threads*: as chamadas de *threads* de espaço de usuário (*user-space*), e as já mencionadas *kernel threads*. A diferença reside em quem possui a responsabilidade pela coordenação de execução dos fluxos: na de espaço de usuário, um programa criado por um usuário é o responsável por essa coordenação, sem a visão global que o sistema operacional possui. Exemplos são o escalonador de Go, que gerencia as *goroutines*, mostradas na seção 3.2.1, e a máquina virtual do Java com o recurso de *green threads*. Já nas *kernel threads*, como o próprio nome indica, o sistema operacional fica responsável por essa gerência. É comum em sistemas UNIX e em Windows essas *threads* serem do padrão POSIX.

2.2 SINCRONIZAÇÃO E COMUNICAÇÃO ENTRE FLUXOS DE EXECUÇÃO INDEPENDENTES

Quando dois ou mais fluxos de execução diferentes executam concorrentemente numa máquina, pode ocorrer de eles requisitarem o acesso a um mesmo recurso (por exemplo, um arquivo, uma tabela, um vetor) ao mesmo tempo. Neste caso, tanto o *output* quanto o *input* dos programas estão comprometidos: uma escrita pode sobrescrever a outra, e a leitura poderia ser feita antes do fim da escrita, ou entre suas execuções, tendo assim

vulnerabilidade a diversos erros de execução. Para lidar com esse problema, é necessário utilizar mecanismos de **sincronização** entre os fluxos de execução para garantir o acesso exclusivo aos recursos, como semáforos e *locks* (STEVENS, 2009). Os recursos de sincronização serão aprofundados na seção 2.2.1.

Outra demanda da computação concorrente é a necessidade de prover mecanismos de **comunicação** entre os fluxos de execução para permitir a troca de dados e requisições entre eles. Esses mecanismos podem ser oferecidos pelo sistema operacional, para comunicação entre processos. No UNIX, por exemplo, é oferecido o enfileiramento de mensagem (POSIX, ou System V), encadeadores de entrada e saída de dados, chamados de *pipes*, e *sockets* de rede, dentre outros.

No contexto de *threads*, as linhas de execução de um processo compartilham o espaço de memória do processo. Da mesma maneira que diferentes processos podem acessar um mesmo arquivo, diferentes *threads* podem acessar uma mesma variável ou área da memória. Esse compartilhamento de memória oferece outra forma de comunicação entre os fluxos de execução, normalmente mais rápida do que a comunicação baseada em troca de mensagens pelo sistema operacional. Por outro lado, impõe uma nova demanda de sincronização no acesso (leitura e escrita) dessas variáveis compartilhadas.

Um exemplo ilustrativo seria um programa que precisa processar dados numéricos e realizar duas operações distintas: pegar o maior dos seus valores e calcular a sua média. No fim, ambos os valores devem escritos na saída padrão, respectivamente o maior valor e a média. Para isso, foi decidido que dois fluxos de execução seriam criados, cada um para uma tarefa. Como não temos garantias sobre a ordem de execução das *threads*, surge a necessidade da comunicação da finalização desses dois fluxos de execução.

O problema apresentado pode ser facilmente resolvido por meio de variáveis globais que indicam a completude das operações: no momento que ambas as variáveis estivessem com um determinado valor, seria determinado que suas execuções completaram, e agora é possível ser feito a impressão dos resultados. A memória compartilhada então, pode ser utilizada para fazer este tipo de comunicação entre as linhas de execução: um fluxo só deve prosseguir dado que o programa atingiu um determinado estado.

2.2.1 Condição de corrida

Imagine que o problema apresentado anteriormente mudou, e agora há o desejo de se criar mais de um fluxo de execução para realizar o cálculo da média, sendo assim calculada mais rapidamente. Os números são somados concorrentemente numa variável global, e no final o valor encontrado é dividido pela quantidade de números somados, por qualquer um dos fluxos. O acesso correto na memória compartilhada não é garantido: duas *threads* podem tentar alterar a mesma variável ao mesmo tempo e o resultado da computação dependerá da ordem em que elas executaram, caracterizando um problema de **condição de corrida**. Isso ocorre porque uma única operação de soma na linguagem de

alto nível, por exemplo, pode ser traduzida em até três instruções de máquina. A ordem de instruções em uma arquitetura x86 executada por cada *thread* pode ser compilada como:

- Ler o valor da memória para um registrador;
- Incrementar em um esse registrador;
- Salvar o valor incrementado na memória;

Assim, para o valor inicial 0, é possível que ambas *threads* executem a instrução de leitura simultaneamente, guardando o mesmo valor 0. Ao incrementar o valor em um o registrador, e depois salvar, o resultado final guardado na memória é 1. O resultado esperado seria 2, em que cada *thread* executaria atonicamente as instruções de ler, incrementar e salvar.

Assim, o resultado do programa passa a ser dependente da ordem que as instruções são executadas no processador. Isso é inaceitável, pois os programas devem ter seu resultado previsível e correto. Para que seja possível eliminar a condição de corrida que surge no acesso à memória compartilhada, sistemas operacionais oferecem, geralmente, dois tipos de ferramentas: trancas (*locks*) e semáforos.

Retornando ao exemplo: toda vez que for realizada um incremento na variável global, é desejado uma garantia de que aquele incremento não será influenciado por outros fluxos, e assim o resultado não seja dependente da ordem das instruções a serem executadas. Para isso, é possível construir uma **seção crítica** do código: antes de realizar o incremento, o fluxo tenta adquirir um *lock*. Como o próprio nome indica, a ideia é proporcionar uma espécie de fechadura, que no momento que um fluxo fechar essa tranca (ou adquire o *lock*), nenhum fluxo conseguirá fazer o mesmo, até que o anterior abra a fechadura (ou libere o *lock*). Assim, temos garantia que, no momento que for realizado o incremento, nenhum outro fluxo estará acessando a variável. A lógica desse código concorrente está demonstrada em 2.1.

Código 2.1 – Pseudocódigo de seção crítica

```
adquirir_lock()
// Inicio se o critica do codigo
soma_global = soma_global + valor_parcial
// Fim da se o critica do codigo
libera_lock()
```

Um outro recurso oferecido é o semáforo binário: a ideia é que, ao invés de adquirir ou liberar *locks*, você incremente/decremente um valor numa estrutura de dado que seja segura concorrentemente. O fluxo, ao chegar na seção crítica, pede para decrementar o valor do semáforo. Se este valor estiver maior que 1, a operação é realizada. Senão,

aguarda-se a mudança do valor 0 para o valor 1. Após a completude da seção crítica, o fluxo deve então incrementar o valor do semáforo, para indicar que o trecho de código está disponível para execução. Apesar de recursos diferentes, a premissa continua a mesma: criar seções críticas do código, e garantir que um único fluxo de execução esteja executando o trecho (BRYANT; RICHARD; RICHARD, 2003).

2.2.2 Sincronização por condição lógica

Voltando ao problema ilustrativo, um último empecilho ficou a ser resolvido: após o final da soma, apenas uma *thread* deve realizar a divisão para o cálculo da média. Assim, um fluxo adquire novamente um *lock* e precisa então descobrir se já foi executada essa operação. Para isso, é utilizado uma espécie de um marcador, que pode ser simples como uma variável global inicializada com o número de *threads* envolvidas no cálculo da média. Após conseguir o acesso a seção crítica, ele decrementa esse valor e verifica se ele chegou a zero. Se sim, a média pode ser calculada por essa *thread*.

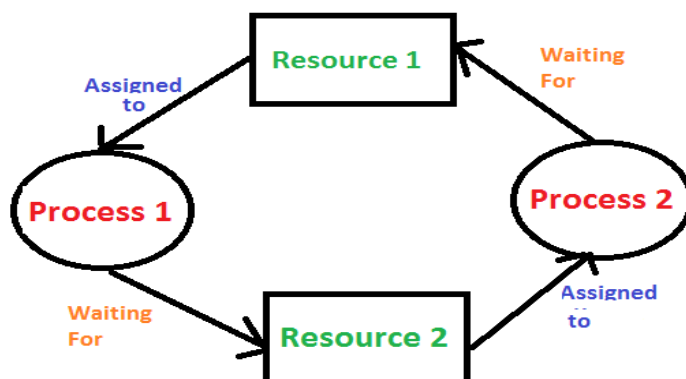
Com esta última lógica, o problema finalmente pode ser considerado resolvido: a soma dos valores não depende mais da ordem de execuções e a média é corretamente calculada no final da execução.

2.2.3 *Deadlocks & Starvations*

Deadlock é um estado que o um determinado conjunto de fluxos de execução pode chegar, em que nenhum consegue avançar o seu processamento, dado que todos esperam de outro fluxo a liberação de um recurso, como por exemplo um *lock*. Genericamente, um programa atinge um estado de *deadlock* por meio de quatro condições (PADUA, 2011):

- Um recurso não deve ser acessado por mais de um fluxo de execução, ou seja, há uma seção crítica no acesso ao recurso;
- Um fluxo de execução utilizando o recurso pode pedir acesso a um outro recurso;
- O recurso só pode ser liberado por meio de ação do próprio fluxo de execução;
- De maneira circular, um fluxo de execução aguarda a liberação de um recurso em uso por outro fluxo, que também aguarda a liberação de um outro recurso do que o primeiro fluxo está utilizando, criando um ciclo de dependências.

É possível observar esse ciclo de dependências na figura 2. Na figura, o processo 1 está aguardando a liberação do recurso 2, e está em posse do recurso 1. Ao mesmo tempo, existe um processo 2, que tem posse do recurso 2 e está aguardando a liberação do recurso 1. Como ambos os processos aguardam liberações que não irão ocorrer, o sistema entra em *deadlock*.

Figura 2 – Condição de *deadlock* entre fluxos de execução

Fonte: Jain, Sandeep. Disponível em <https://www.geeksforgeeks.org/operating-system-process-management-deadlock-introduction/>. Acesso em 02 jul.2019.

Deadlock nunca é um estado desejado para o programa, simboliza um erro de sincronização de recursos, e é um dos desafios da área de concorrência. Programas complexos e extensos, com múltiplas dependências de acesso e múltiplos fluxos executando incrementam a chance de ocorrer esses estados errôneos.

Diferentemente, *starvation* é um estado possível para o programa, apesar de comumente indesejado, em que algum fluxo não consegue acesso a um determinado recurso por um tempo muito longo.

Existem soluções de problemas que permitem o *starvation* de um determinado fluxo, como mostrado na seção 4.1.2 de leitores e escritores, no capítulo 4. Nestes cenários, a ideia é que há uma prioridade maior para um determinado tipo de fluxo, que sempre que possível deve receber acesso ao recurso, pagando o preço de impedir o acesso por outro fluxo de execução.

3 A LINGUAGEM DE PROGRAMAÇÃO GO

Go (comumente chamado de *Golang*, abreviação de *Go programming language*) é uma linguagem de programação de código aberto, lançada em 2009 pela Google. Foi criada por Robert Griesemer, Rob Pike e Ken Thompson, sendo Thompson também participante na criação dos sistemas Unix e da linguagem de programação C. O começo de seu desenvolvimento se deu em 2007, dois anos antes do lançamento a público (PIKE et al., 2019a). O alto tempo de compilação e a dificuldade de manutenção de código C++ para os sistemas de alto desempenho motivaram a criação da linguagem. É uma linguagem compilada com *garbage collector*, com uma sintaxe similar a C/C++, porém com mecanismos mais sofisticados, como funções de primeira classe, *closures*, e um modelo similar ao de orientação de objeto por meio de *interfaces* (BINET, 2018).

Go, criada já na era de máquinas de múltiplos processadores, é uma linguagem concorrente, em especial a algumas primitivas diferenciais da ferramenta: *goroutines*, semelhante a *green threads* do Java (porém com paralelismo), permitindo que funções sejam invocadas de maneira concorrente através da diretiva *go*; *channels*, utilizada para fazer a comunicação entre *goroutines*; e *select*, utilizado para o controle de fluxo em códigos concorrentes.

Além dessas primitivas, Go disponibiliza também uma biblioteca básica de concorrência denominada *sync*, que oferece desde mecanismos clássicos de sincronização como *mutexes*, até estruturas mais complexas, como um *HashMap* compartilhado com acesso seguro entre *goroutines*.

3.1 ESPECIFICAÇÕES

Go é uma linguagem compilada, com tipagem estática e recursos para tipagem dinâmica. A linguagem se assemelha a C, porém injeta recursos novos de linguagens modernas, com o propósito de simplificação de código.

Código 3.1 – Código “olá mundo” em Go

```
package main
import "fmt"
func main() {
    fmt.Println("Hello World!")
}
```

3.1.1 Packages

Semelhante a linguagens orientadas a objetos, a linguagem Go utiliza o recurso de pacotes (*packages*). Como visto no código 3.1, na primeira linha de cada código, deve-se

explicitar qual é o pacote que o código pertence. A visibilidade de uma função ou de uma estrutura é definida pela capitalização do primeiro carácter de uma variável. Ou seja, caso a primeira letra do nome da variável seja maiúscula, significa que ela é visível fora do seu pacote.

Os pacotes, por padrão, possuem nomes todos minúsculos, e sua importação deve ser feita no início do programa, imediatamente após a definição de qual pacote aquele arquivo pertence.

O *main package*, ou pacote principal, deve conter uma função chamada `main`, que define o início do fluxo do programa.

3.1.2 Tipos de dados

Existem duas maneiras de declarar variáveis. Utiliza-se o operador `:=` para declaração de variáveis implícitas, onde o tipo da nova variável é inferido a partir do resultado da expressão a direita. Já quando deseja-se declarar o tipo diretamente, deve ser utilizado a palavra reservada `var`, seguida do nome das variáveis, e por fim o tipo de dado daquelas variáveis.

A linguagem possui já pré declarado uma vasta gama de tipos de variáveis (PIKE et al.,):

- Tipos booleanos: `bool`, representa o valor `true` ou `false`, para operações de lógica booleana. O código A.1 declara uma variável por meio do operador `:=` com o valor `true`, e imprime na tela.
- Tipos numéricos: `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32`, `int64`, `float32`, `float64`, `complex64`, `complex128`. Cada tipo representa se é um `unsigned integer`, `integer`, `float` ou `complex`, com a quantidade de bits utilizados para construir a variável. No código A.2, temos exemplos de declaração das variáveis por meio da palavra reservada `var`, e exemplos de conversão de tipos para realizar as operações entre tipos diferentes de dados. Nesse código, há exemplos também de atribuições por meio do operador de atribuição `=`, que apenas atribui valor a uma variável já existente.
- Tipos para texto: o tipo `string` guarda um conjunto de bytes. *Strings* em Go são imutáveis. No código A.3, temos um exemplo de operação de soma entre *strings*. A *string* final, resultado das strings salvas nas variáveis `a` e `b` é uma nova string, e não uma modificação das anteriores.
- *Arrays*: são sequências de elementos que compartilham o mesmo tipo. Podem ser multidimensionais.

- *Slices*: são uma descrição para algum tipo de *array* existente, provendo acesso a uma parte do *i*, podendo ser ou não todo o *array*. Um *slice* não inicializado tem sempre valor *nil*. Um *slice* não tem de fato um valor salvo na memória: ele apenas referência um *array*, que de fato guarda o valor linearmente na memória. O código A.4 exemplifica a diferença entre um *array* e um *slice*. A variável de nome *odds* grava os cinco números ímpares entre 0 e 10. É um *array* de tamanho 5. Já a variável de nome *firstThreeOdds* pega os valores dos 3 primeiros campos da variável *odds*. A regra de acesso entre colchetes é inclusiva pro primeiro índice e exclusivo pro último índice. Como a contagem começa no 0, *odds*[0:3] significa os elementos de índice 0, 1 e 2 do *array odds*.
- *Structs*: *Structs*, ou estruturas, criam novos tipos de dados, que possuem campos com nome e tipos definidos a priori. Estruturas e interfaces devem ser declaradas indicando a criação de um novo tipo por meio da palavra reservada **type**. No código A.5, é criado o tipo de dado *foo*, que possui o campo *Bar* que é uma *string*, e o campo *Baz* que é um inteiro.
- Ponteiros: semelhante a um ponteiro em C, guarda o local de memória de uma variável do tipo base do ponteiro. Um ponteiro não inicializado tem valor *nil*. No código A.5, temos uma declaração de um ponteiro para a estrutura *foo*, por meio do operador **&**.
- Funções: Uma variável do tipo função que possui o mesmo conjunto de parâmetros e resultados que o definido na variável. O último parâmetro pode receber uma quantidade não definida, sendo assim uma *variadic function*. São definidas por meio da palavra reservada **func**. No código A.5 é possível ver um exemplo de uma função de multiplicação, que pega dois valores passados por parâmetro e retorna o resultado da multiplicação deles.
- Métodos: Um método é uma função que está associada a uma determinada estrutura. O método pode ser tanto associado a uma estrutura quanto a um ponteiro para uma estrutura.
- Interfaces: Semelhante às interfaces das linguagens com orientação a objeto, define uma lista de métodos que uma estrutura deve implementar. Assim, para afirmar que uma estrutura *X* implementa uma interface *Y*, é necessário que a estrutura *X* implemente todos os métodos definidos pela interface *Y*. No código A.6, temos o exemplo de uma interface *Vec*. Note que ela descreve um novo tipo de dado. A interface *Vec* possui dois métodos: um de soma, e um de subtração, em que cada um recebe um argumento do tipo *Vec* e também um *Vec*. A estrutura de dados *Vector2* implementa todos os métodos da interface *Vec*. Assim, na função principal,

é possível fazer uma atribuição da variável `v1` do tipo `Vec` para uma estrutura do tipo `Vector2`. Antes da atribuição, o valor da variável `v1` é nulo. Uma variável cujo tipo é uma interface qualquer, possui ou o valor de uma estrutura real que implementa esta interface, ou o valor nulo.

- Mapas: Um tipo de dicionário chave-valor não ordenado que associa um determinado tipo de variável (a chave) a outro (o valor). O tipo da chave e o tipo do valor devem ser especificados anteriormente, inclusive um mapa pode ser usado como o tipo de uma chave ou de um valor. O código A.7 exemplifica o uso de um mapa onde tanto as chaves quanto os valores são textos.
- Canais: Estruturas de comunicação entre códigos concorrentes que possui duas operações: envio e recebimento. É utilizado para troca de mensagens entre *goroutines*.

3.1.3 Controles de fluxo

O controle de fluxo em Go é feito por meio estruturas condicionais ou de repetição. Para diminuir a redundância e simplificar a maneira de escrever um código, existe apenas uma única estrutura de repetição, que usa a palavra reservada `for`. Para criação dos fluxos condicionais, existem os mecanismos de `if...else`, `switch` e o `select`, sendo o último utilizado para controle de fluxo em códigos concorrentes.

O `if` sempre inclui uma expressão condicional que deve retornar um valor booleano, e será avaliada para entrar ou não no fluxo. O comando pode conter uma declaração anterior a sua condicional, separada por meio de um ponto e vírgula, e deve também sempre se utilizar chaves para definir o escopo do fluxo. Essa declaração serve para permitir que uma variável pertença apenas ao fluxo, tanto quando o retorno é *true*, fluxo do *if*, ou quando é *false*, no fluxo do *else*. Caso seja necessário utilizar também o `else`, que executa quando a condição não é verdadeira, este deve estar exatamente após a chave que fecha o fluxo do `if`.

No código A.8 temos que, para o valor inicial de `a = 10`, as impressões feitas serão `"a' maior que 5"` e `"a/2' menor que 10"`. A nova variável criada na declaração dentro da cláusula condicional só existe dentro do fluxo do `if`, como o caso da variável `b` do exemplo..

O `switch` é um tipo de controle de fluxo para múltiplas escolhas. Deve-se passar uma expressão, e cada possível fluxo a ser seguido é definido por uma expressão precedido da palavra `case`. Caso deseje-se um comportamento padrão, utiliza-se o caso `default`, que é executado quando a variável não corresponde a nenhuma das expressões de cada caso. As expressões de caso devem poder executar a comparação `v == t`, onde `v` é a expressão inicial do `switch` e `t` é a expressão do caso.

Para controle de repetição, existe o mecanismo do `for`. O `for` segue uma sintaxe semelhante a de C, porém sem a necessidade de parênteses. Após a palavra reservada `for`, deve-se ter os comandos a serem executados antes do início da repetição. Depois, a condição de parada, e por fim o que deve ser executado no fim de cada passo do *loop*. Usa-se o ponto e vírgula para separação dessas declarações. O trecho de código apresentado no código A.9 ilustra um exemplo básico do uso do `for`.

O código A.9 pode ser feito de maneira mais simples, por meio de um *for* := *range*, recurso para iterar sobre um determinado tipo de dado que pode ser do tipo *map*, *array*, *slice* ou *channel*. Com mapas, *arrays* e *slices*, é retornado dois valores: uma chave e um valor, onde a chave é o índice no caso de *arrays*, *slices* ou *strings*, e para *maps* é a variável definida como chave do mapa. No caso de canais, só é retornado um único valor, que é o valor recebido pelo canal, e o *loop* só é finalizado quando é feito o fechamento do canal. O código A.10 mostra como ficaria o programa A.9 utilizando o `range`.

3.1.4 Interfaces e Métodos

Uma interface pode ser definida como um “protótipo”, ou um tipo não definido, em que a intenção de uso é para abstrair qual é a estrutura de uma variável, apenas definindo quais são os métodos que ela disponibiliza.

Um bom uso de interfaces é para abstrair a comunicação entre pacotes diferentes de um mesmo sistema (por exemplo, um pacote que acessa o banco de dados), ou com outros pacotes externos (por exemplo, um pacote obtido na internet que implementa gRPC). Um exemplo é mostrado no código A.11, no qual o programa envia logs de processos para um servidor remoto, podendo ser tanto via UDP quanto por TCP, dependendo de uma *flag* em sua inicialização. Nesse caso, deseja-se que o programa execute a mesma lógica em qualquer um dos casos, sendo indiferente qual é protocolo da camada de transporte.

Na função `SendStringToConn`, no código A.11, não se sabe qual o tipo de conexão será feita (TCP ou UDP). Como falado na subseção 3.1.2, basta que seja uma estrutura de dado qualquer que implemente os métodos definido pela interface para o Go reconhecer que aquela estrutura é um tipo válido para a interface. Assim, basta implementar os métodos `Read` e `Write` com os mesmos parâmetros e retornos definidos pela interface `Connection`.

A implementação de um método para uma estrutura é feita de forma semelhante à implementação de uma função, porém ela é precedida por um ponteiro ou uma variável do tipo da estrutura a que esse método pertence. O nome desse campo é receptor do método, e por meio da variável é possível acessar os campos da estrutura dentro do método. Os métodos implementados foram o `Read` e `Write` para a estrutura `TCP`, necessários para garantir que essa estrutura implementa a interface `Connection`.

3.2 RECURSOS PARA CONCORRÊNCIA

Go foi desenhada de maneira a facilitar a escrita de códigos concorrentes. Para isso, foram criados alguns recursos que cuidam da criação, comunicação e controle de fluxos de execução concorrentes. Serão apresentados a seguir os principais recursos oferecidos pela linguagem: *goroutines*, *channels*, *select* e a biblioteca *sync*.

3.2.1 Goroutines

Goroutines são um tipo de *user space threads*, como visto na seção 2.1, gerenciadas pelo escalonador do Go. Para invocar uma *goroutine* basta fazer a chamada de uma função qualquer precedida pela diretiva *go*. Com um tamanho reduzido de inicialmente 2KBs, a criação de uma *goroutine* é leve e não custosa (em comparação com uma *kernel thread* de 2MB na arquitetura x86_64 (KERRISK, 2018)).

Por funções serem valores de primeira classe, a criação do fluxo concorrente via *goroutine* pode vir de uma função anônima, como ilustrado no código 3.2.

Código 3.2 – Exemplo de função anônima concorrente

```
package main

import (
    "fmt"
    "time"
)

func processData() {
    // Pega dados de um servidor remoto e processa-os
}

func main() {
    go func() {
        for {
            fmt.Println("working...")
            time.Sleep(30 * time.Second)
        }
    }()
    processData()
}
```

É possível ver que no código 3.2 iniciamos uma *goroutine* que executa uma função anônima que entra num loop sem condição de parada, imprimindo na linha de comando um texto para dar alguma mensagem para o usuário (no caso, que está trabalhando). O que encerra a execução da *goroutine* é a finalização da *main thread*, o fluxo principal da função *main*. Esse comportamento é semelhante ao encontrado numa *kernel thread*,

em que o fim da execução da *thread* principal implica no fim do processo. Quando um processo é finalizado, todas as suas *threads* são finalizadas também.

Go adota um modelo de **gerência cooperativa de tarefas** em que o escalonador só permite a execução de uma goroutine de cada vez para cada *kernel thread* disponível. Isso significa que não existe preempção entre *goroutines* que executam em uma mesma *kernel thread*. Pode-se controlar a quantidade de *kernel threads* a serem criadas a partir da variável de ambiente GOMAXPROCS. Por padrão, o valor dela é exatamente a quantidade de processadores disponíveis na máquina.

Dessa maneira, no código 3.2, caso haja apenas um único núcleo de processamento disponível na máquina e decidimos executar o programa com a variável de ambiente GOMAXPROCS com valor 1 (definindo o número máximo de *kernel threads* igual a 1), a execução da *goroutine* só será possível quando ocorrer algum tipo de parada na função `processData`, como ler de um canal, escrever no canal, esperar IO do sistema operacional, entre outros. Caso fosse especificado um número maior que 1 de *kernel threads* com a variável GOMAXPROCS, a *goroutine* teria então espaço para ser alocada. Quem é responsável por gerenciar esse tipo de *thread* é o próprio sistema operacional, então o código iria executar concorrentemente a partir das preempções de processos/*threads*, como no modelo convencional de *threads* POSIX, visto em 2.1.

Goroutines, apesar do seu nome, não se comportam como corrotinas. Não é possível cessar sua execução e retomar do mesmo ponto por mecanismos disponíveis nativamente pela linguagem, sendo necessário assim uma implementação de corrotinas utilizando os recursos do Go. No próximo capítulo, um exemplo implementação de corrotinas assimétricas em Go será demonstrado.

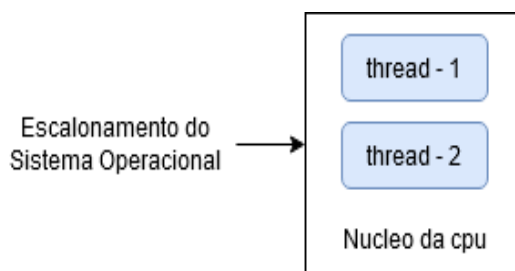
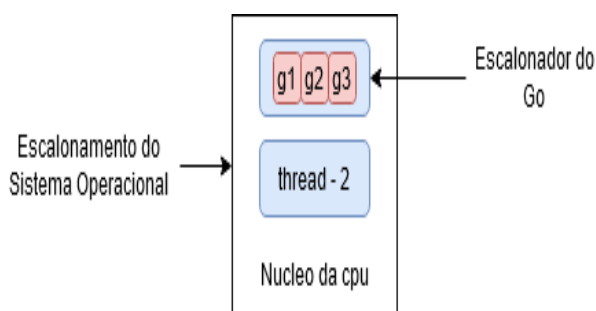
3.2.1.1 Escalonamento de *goroutines*

Goroutines não são mapeadas de maneira um para um em *threads* de sistemas operacionais, e consequentemente não são escalonadas automaticamente para sua execução. O *runtime* do Go (programa auxiliar de execução de códigos Go) possui um algoritmo próprio de escalonamento (JOSHI, 2018). Na figura 3 mostra-se comparativamente as diferenças entre o escalonamento de *threads* comuns e o escalonamento de *goroutines* na figura 4.

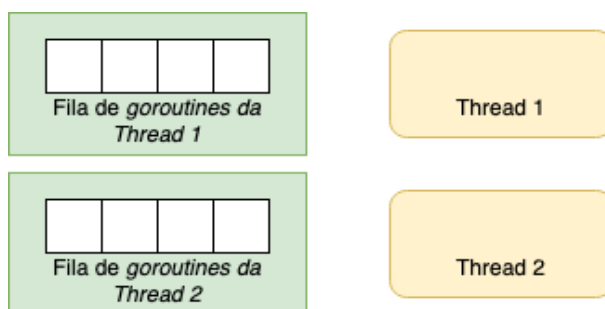
O algoritmo funciona da seguinte maneira: para cada *kernel thread* criada, até o número máximo definido pela variável GOMAXPROCS, cria-se uma fila de *goroutines* que serão executados nessa *kernel thread*. A cada nova *goroutine* que chega, avalia-se se existe uma *kernel thread* disponível ou se pode criar uma nova. Se não houver espaço, ela é colocada numa fila seguindo uma ordem circular.

Num cenário onde apenas temos duas *kernel threads*, a fila começaria como indicado na figura 5. Ao iniciar a primeira *goroutine* (a *goroutine* principal, por exemplo), aloca-se ela numa fila qualquer com menor tamanho, como indicado na figura 6. Caso a *thread*

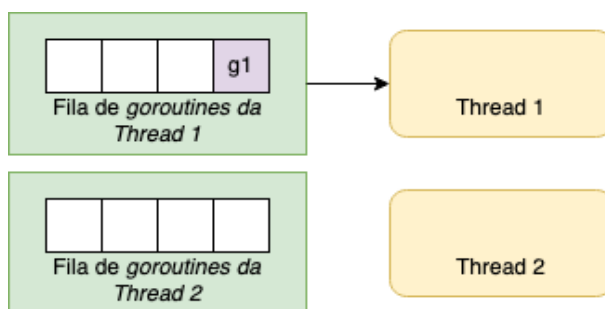
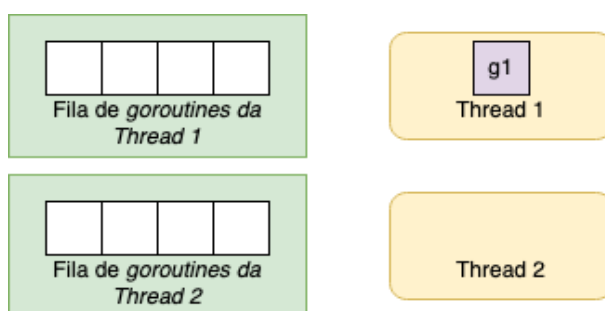
Figura 3 – Escalonamento do sistema operacional

Figura 4 – Escalonamento das *goroutines* dentro de *threads* do sistema operacional

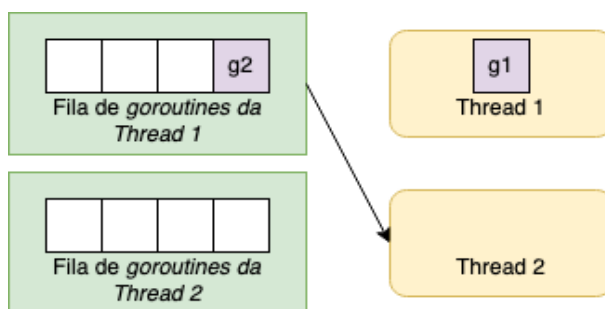
não esteja executando nada, ela é "desestacionada", ou *unparked*, e inicia-se a execução da goroutine retirada da fila, como indicado na figura 7.

Figura 5 – Fila de goroutines com duas *kernel threads*

Após a chegada de uma nova *goroutine*, novamente coloca-se numa fila qualquer de menor tamanho. Para permitir um aproveitamento máximo das *threads* de sistema operacional, essas filas são preemptivas: uma *goroutine* aguardando liberação da *kernel thread* 1 pode ser movida para a fila da *kernel thread* 2, caso esta outra *thread* esteja disponível para utilização. Assim, na figura 8, a *goroutine* 2 é alocada para uma *kernel thread* diferente da que a sua fila pertencia, sendo assim "roubada" por uma outra fila. Este esquema é conhecido como *job stealing*, em que cada unidade de processamento ociosa (neste caso,

Figura 6 – Chegada de uma *goroutine*Figura 7 – Alocação da *goroutine* 1

a *thread*) rouba trabalho alocado para outra unidade de processamento, como vista na figura 9.

Figura 8 – Chegada de uma nova *goroutine*

Por fim, novas *goroutines* aguardam pelo fim ou por uma ação bloqueante (como ler de um canal) das *goroutines* sendo executadas para poderem ter sua vez de execução, como indicado pela figura 10.

Figura 9 – *Job stealing*: *goroutine* é alocada numa *kernel thread* diferente da fila a qual pertencia

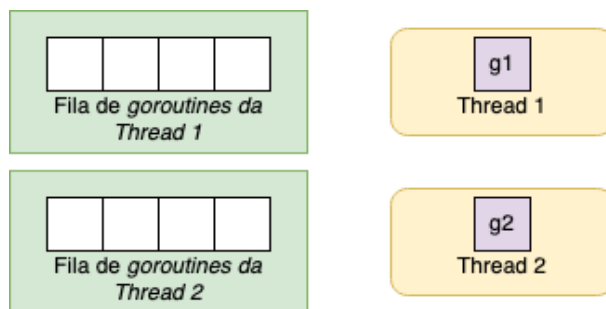
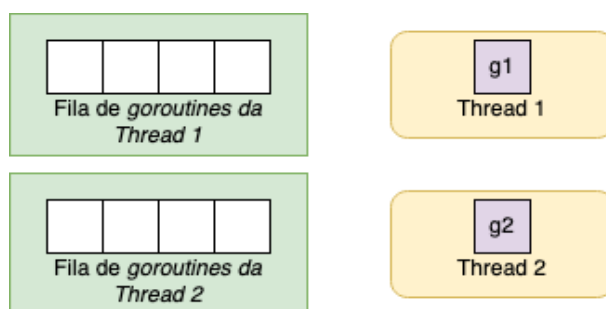


Figura 10 – Novas *goroutines* criadas



3.2.2 Channels

Enquanto *goroutines* são o mecanismo de criação de diferentes fluxos de execução, *channels* (ou canais) são o mecanismo de comunicação entre esses diversos fluxos. Existem dois tipos de *channels*, buferizados e não buferizados, sendo os não buferizados bloqueantes e os buferizados não bloqueantes (enquanto houver espaço no buffer para o envio, ou se tiver algo preenchido no buffer para recebimento).

Channels são criados utilizando o comando **make**, que aloca um determinado tipo de dado na memória e retorna a sua referência. Como a linguagem possui um *garbage collector*, não é necessário a liberação de memória explícita. *Channels* oferecem dois tipos de operações: envios e recebimentos (semelhante numa comunicação interprocessos). Utiliza-se o operador **<-** para executar estas operações. No código 3.3, a função principal tenta fazer uma leitura do canal não buferizado criado, enquanto a *goroutine* criada realiza uma escrita. Para criar um canal buferizado, deve-se passar como o segundo argumento da função **make** o tamanho do *buffer*.

Código 3.3 – Exemplo simples de um canal

```
package main

import "fmt"

func main() {
    ch := make(chan string)
    go func() {
        ch <- "Hello world!"
    }()
    a := <-ch
    fmt.Println(a)
}
```

3.2.3 A biblioteca *sync*

Quando a modelagem por troca de mensagem é difícil e complexa, Go oferece também os recursos clássicos, como *mutexes* e semáforos por meio da biblioteca **sync**. Além disso, outras estruturas são oferecidas para serem usados de maneiras complementares aos *channels* e *goroutines*.

Utilizaremos o recurso de sincronia da biblioteca **sync** chamado **WaitGroup**, um contador seguro entre *goroutines*, utilizado comumente para realizar a espera pela conclusão dos fluxos concorrentes (ideia similar a um mecanismo de sincronização por barreira). Ele possui um contador interno iniciado em 0 e três funções para interagir com ele: **Add**, que soma um número qualquer ao contador, **Done** que decrementa em uma unidade o contador, e por fim o **Wait**, que trava a *goroutine* se o valor do contador interno for diferente de 0.

Código 3.4 – Comunicando por canais

```

package main

import (
    "fmt"
    "sync"
    "net/http"
)

func main() {
    downloadSuccess := make(chan bool)
    wg := sync.WaitGroup{}
    wg.Add(1)
    go func() {
        defer wg.Done()
        if <-downloadSuccess == true {
            fmt.Println("Download success")
        } else {
            fmt.Println("Download failed")
        }
    }()
    wg.Add(1)
    go func() {
        defer wg.Done()
        _, err := http.Get("http://example.com")
        if err != nil {
            downloadSuccess <- false
            return
        }
        downloadSuccess <- true
    }()
    wg.Wait()
}

```

No código 3.4, tem-se um programa que cria duas *goroutines* e aguarda o fim da execução das duas por meio do `wg.Wait()`. A primeira *goroutine* checa o valor de um canal e imprime uma mensagem de acordo com o resultado da leitura. A segunda *goroutine* preenche o canal a partir do resultado de um *download* de uma página de exemplo da Internet. O fluxo principal fica responsável por criar o canal booleano compartilhado por todas as *goroutines*. Lembrando que, as invocações das *goroutines* não iniciam sua execução de fato, caso não haja *kernel threads* disponíveis, devido a sua propriedade cooperativa. Assim, por exemplo, numa execução com `GOMAXPROCS=1` e, conseqüentemente, tendo apenas uma *thread* de *kernel*, apenas na chamada do `wg.Wait` o escalonador de Go é ativado para iniciar a execução de alguma das outras duas *goroutines*.

Usa-se o comando **defer** para “agendar” a execução de uma função para o fim da função onde foi chamado, mesmo se ocorrer algum erro na função. Utiliza-se esse comando no código 3.4 para garantir que o contador do **WaitGroup** será incrementado.

Caso a primeira venha a ser executada, ela tentará ler do canal **downloadSuccess**. Como o canal ainda não foi preenchido, ela então libera a sua execução e fica numa espera. Assim, a outra *goroutine* pode então executar e preencher o canal com **true** caso nenhum erro no download seja encontrado.

Se a segunda *goroutine* fosse executada antes da primeira, ao tentar enviar no canal, não haveria nenhuma *goroutine* executando uma operação de recebimento, o que travaria sua execução e liberaria a *kernel thread* para outra *goroutine*.

O exemplo 3.4 demonstra um bom uso da comunicação de dados em Go. Muitas vezes a intenção de compartilhamento de memória é para permitir que diversos fluxos de execução fiquem observando uma mudança. Esse tipo de observação, normalmente causa espera ocupada, onde é gasto ciclos de instruções numa *thread* para que ela veja se o valor de uma variável foi modificado ou não. Neste modelo, não há espera ocupada: a *goroutine* só está executando enquanto há algum comando a ser executado, e libera a execução para outras *goroutines* quando é feito algum tipo de bloqueio, como no caso a leitura de um canal não buferizado.

3.2.4 Select

Select é um mecanismo de controle de fluxos concorrentes. O comportamento é similar a de um *switch*, que permite um fluxo condicional de múltiplas opções. A diferença é que, no *select*, as condições são operações de escrita ou de leituras em um canal qualquer.

Como também no *switch*, é possível escolher um comportamento padrão, a partir do caso especial **default**, sendo escolhido se as operações de todos os casos resultarem num bloqueio. No caso de canal buferizado significa que não possui nenhum valor no buffer no caso da leitura, ou não possui espaço em caso da escrita. Para o canal não buferizado, significa que não há nenhuma outra *goroutine* para receber o dado caso a condição seja um envio, ou nenhuma outra *goroutine* para enviar o dado caso a condição seja um recebimento.

O *select* permite a escrita de fluxos comuns em códigos sequencias clássicos, adicionando a possibilidade de decisões de múltiplos fluxos em códigos concorrentes. Semelhante ao *switch*, permite também um comportamento padrão caso nenhuma outra condição seja cumprida.

Código 3.5 – Exemplo de uso do select

```

package main

func getLastTemperatureValue() float32 {
    // reads from sensor
}

func getLastMoistureValue() float32 {
    // reads from sensor
}

func processTemperature(temp float32) {
    // send to server
}

func processMoisture(moisture float32) {
    // send to server
}

func readTemperature(temperatureCh chan float32) {
    // ...
    for {
        temperatureCh <- getLastTemperatureValue() //
    }
}

func readMoisture(moistureCh chan float32) {
    // ...
    for {
        moistureCh <- getLastMoistureValue() //
    }
}

func main() {
    tempChannel := make(chan float32)
    moistureChannel := make(chan float32)
    go readMoisture(moistureChannel)
    go readTemperature(tempChannel)
    for {
        select {
        case temp := <-tempChannel:
            processTemperature(temp) //
        case moisture := <-moistureChannel:
            processMoisture(moisture) //
        }
    }
}

```

No código 3.5, mostra-se um possível cenário em que, um servidor central recebe diversos sinais de sensores, caracterizados pelas funções `getLastTemperatureValue()` e `getLastMoistureValue()`, e os processa sem uma ordem definida. Pela definição da declaração `select`, apenas um único caso será executado de cada vez (PIKE et al.,). Isso significa que ainda é possível ocorrer *starvations*, dado o comportamento pseudo-aleatório da escolha caso os dois casos estejam prontos.

Um exemplo de uso do `select` é num cenário onde esperamos receber dados de múltiplas fontes, porém sem uma ordem definida. Além disso, o `select` pode ser utilizado de forma não bloqueante. O recebimento ou envio de uma mensagem num canal é primeiramente validado se é possível de ocorrer. Caso não seja possível enviar para ou receber de um canal, o código pode seguir um fluxo padrão e terminar a execução do trecho. No código 3.6, o `select` irá selecionar algum dos canais, caso o envio já tenha sido concluído. Se nenhuma outra *goroutine* estiver enviando no canal, ele apenas segue o padrão, e finaliza o programa.

Código 3.6 – Exemplo de `select` com leitura não bloqueante

```
package main

import "fmt"

func main() {
    ch, ch2 := make(chan int), make(chan int)
    go func() {
        ch <- 10
    }()
    go func() {
        ch2 <- 20
    }()
    select {
    case n := <-ch:
        fmt.Printf("Received value %d\n", n)
    case n := <-ch2:
        fmt.Printf("Received value %d\n", n)
    default:
        fmt.Println("Couldn't read from any channel")
    }
}
```

4 USO DOS RECURSOS DE CONCORRÊNCIA DE GO

Neste capítulo são apresentadas propostas de resoluções de problemas clássicos de concorrência, utilizando os recursos de concorrência oferecidos por Go. São mostradas também propostas de soluções para problemas de sistemas distribuídos: uma aplicação no modelo clássico de cliente/servidor; e uma aplicação simples de mensagem instantânea, ou *chat*. Em seguida, apresenta-se uma proposta de implementação de corrotina para Go. Por fim, uma análise simplificada de desempenho da linguagem é apresentada, comparando os resultados de uma solução para o problema de multiplicação de matrizes em Go e C+++. Todos os programas desenvolvidos estão disponíveis também no Github (GUIO, 2019).

4.1 PROBLEMAS CLÁSSICOS DE CONCORRÊNCIA

Nesta seção serão abordados três problemas clássicos de concorrência: produtores e consumidores, leitores e escritores e o barbeiro dorminhoco.

4.1.1 Produtor e consumidor

O problema de produtor-consumidor, também conhecido como o problema do fluxo de dados compartilhados (*buffer*) limitado, foi proposto por Dijkstra como um problema de sincronização do uso de recursos compartilhados (DIJKSTRA, 1972). Para um *buffer* qualquer, há duas entidades que alteram este *buffer*: produtores devem inserir porções de dados do *buffer* e consumidores devem retirar as porções de dados presentes no *buffer*, na mesma ordem que foram inseridos e sem repetição no consumo. Além disso, as informações inseridas no *buffer* não podem ser perdidas.

4.1.1.1 Um produtor e um consumidor com *buffer* de tamanho mínimo

No cenário de um único produtor e um único consumidor, o problema de sincronização de acesso é comumente resolvido por meio da criação de uma seção crítica para verificar o estado correto do *buffer* e modificá-lo atômicamente. A premissa é que um fluxo de execução produtor deve inserir no *buffer* caso haja espaço livre. Caso não exista espaço livre no *buffer*, ele deve ser bloqueado e aguardar o consumidor retirar valores do *buffer*. Já o consumidor, deve tentar retirar o próximo valor do *buffer*, e caso não exista elementos no *buffer* ele deve ser bloqueado e aguardar o produtor inserir valores no *buffer*.

Em Go, a abordagem utilizada é diferente. Como visto na subseção 3.2.2, a linguagem Go provê um recurso de sincronização baseado num modelo de troca de mensagens, os canais. Dado as propriedades de garantia de envio e recebimento de um canal de maneira síncrona e ordenada (os valores de um canal qualquer são retirados na mesma ordem em

que foram inseridos), o problema do produtor consumidor pode ser resolvido trivialmente utilizando um canal.

Código 4.1 – Código do produtor com apenas um consumidor e um produtor

```
package main

import (
    "fmt"
)

func producer(stream chan string) {
    var str string
    for {
        fmt.Scanln(&str)
        stream <- str
    }
}
```

No código 4.1, é declarado uma variável `str` que recebe o valor lido da entrada padrão. Após a leitura, esse valor é enviado no canal. Se não tiver espaço no *buffer*, o envio não é concluído, e aguarda-se a liberação de espaço para prosseguir.

Código 4.2 – Código do consumidor com apenas um consumidor e um produtor

```
package main

import (
    "fmt"
)

func consumer(stream chan string) {
    for {
        str := <-stream
        fmt.Println("Value received:", str)
    }
}
```

No código 4.2, apenas declara-se a variável que receberá o valor do canal, e realiza-se a operação de recebimento neste canal. Após o recebimento, é feita a impressão na tela. Se não houver valor no canal, a execução dessa *goroutine* é interrompida até a inserção de algum valor no canal.

Código 4.3 – Código da função principal com apenas um consumidor e um produtor

```
package main

func main() {
    stream := make(chan string, 1)
    go consumer(stream)
    go producer(stream)
    select {}
}
```

Para o código 4.3, resta então a função principal que inicializa os parâmetros que serão utilizados nas funções. Ele cria o canal que será compartilhado pelas *goroutines*, nesse caso como um buffer de um único slot. Para impedir o fim da execução do programa após a criação das *goroutines*, utiliza-se um `select` sem nenhuma cláusula.

O código é suficiente para suprir o seguinte cenário: dado um produtor e um consumidor, o produtor gera uma nova *string* a partir da entrada padrão e a insere num *buffer* que só aceita uma única sequência de *bytes* por vez, e o consumidor obterá a sequência inserida. Não importa a ordem que o código seja executado, o resultado é sempre o mesmo, dado que o consumidor espera o produtor caso este ainda não tenha produzido nada, ao mesmo tempo que o produtor aguarda o consumidor enquanto o *buffer* estiver lotado.

4.1.1.2 *N* produtores e *M* consumidores, com *buffer* de tamanho qualquer

Para este tipo de cenário, uma solução clássica é utilizar novamente seções críticas para a inserção e remoção dos dados do *buffer*. Porém, dado que o *buffer* tem um tamanho qualquer, é necessário controlar qual é a posição no *buffer* que deve receber o próximo valor inserido, e qual a posição no *buffer* para o próximo valor retirado. Assim, alguma estrutura de fila é necessária.

Em Go, não há absoluta mudança nas implementações dos produtores e consumidores. Dado que a leitura e recebimento em um canal já é ordenada e síncrona, o problema já se encontra resolvido. Assim, a alteração é realizada somente no código da função principal, para inicializar o canal com um valor qualquer de uma variável, e também inicializar mais de um fluxo de execução dos produtores e consumidores, como ilustrado no código 4.4

Código 4.4 – Código da função principal com N consumidores e M produtores

```
package main

func main() {
    consumers := 100 // valor qualquer
    producers := 100 // valor qualquer
    bufferSize := 1000 // valor qualquer

    stream := make(chan string, bufferSize)

    for i := 0; i < consumers; i++ {
        go consumer(stream)
    }
    for i := 0; i < producers; i++ {
        go producer(stream)
    }
    select {}
}
```

Dado as características das estruturas de dados de canais de Go, a solução do problema do produtor consumidor torna-se trivial. Não é necessário criar seções críticas no código, as funções produtoras e consumidoras podem ser executados em uma ordem qualquer, dado que a estrutura de canal provê um método seguro e ordenado de troca de mensagens por meio de memória compartilhada.

4.1.2 Leitores e escritores

O problema dos leitores e escritores pode ser definido da seguinte maneira: um recurso é compartilhado por diversos fluxos de execução que precisam escrever ou ler este recurso, de maneira exclusiva (BALLHAUSEN, 2003).

O problema possui uma característica importante diferente do problema do produtor consumidor: vários leitores podem realizar uma leitura simultaneamente, enquanto apenas um escritor pode realizar a escrita por vez, considerando que os tempos de leitura e escritas são tempos finitos. Não há, então, uma maneira trivial de modelar este tipo de problema por meio de trocas de mensagens: o mesmo recurso pode ser lido inúmeras vezes, ao mesmo tempo que pode ser escrito sem nunca ser lido.

Serão consideradas três implementações do problema:

- Implementação básica, cumprindo os requisitos básicos do sistema.
- Implementação com prioridade para a escrita;
- Implementação sem prioridade (também conhecido como sem *starvation*).

4.1.2.1 Implementação básica

Na implementação básica do problema leitores escritores, temos os seguintes cenários:

- Caso um processo **leitor** requisite o acesso ao recurso, ele irá:
 - Ler o recurso, caso não exista nenhum escritor acessando o recurso ou;
 - Aguardar a liberação do recurso, caso um escritor esteja utilizando o recurso.
- Caso um processo **escritor** requisite o acesso ao recurso, ele irá:
 - Escrever no recurso, caso não tenha nenhum escritor nem nenhum leitor utilizando o recurso, ou;
 - Aguardar a liberação do recurso, caso tenha algum outro processo utilizando esse recurso. No caso dos leitores, ele aguarda o fim de todos os leitores.

Nessas condições não há uma garantia de tempo para os escritores dado que, se ao menos um leitor consegue acesso ao recurso, novos leitores sempre poderão acessar o recurso, podendo então causar *starvation* nos escritores.

Código 4.5 – Código do escritor para implementação básica

```
package main

import (
    "sync"
)

type Writer struct {
    ID      int
    Writers *sync.Mutex
}

func (w *Writer) start() {
    for {
        w.Writers.Lock()
        // escreve no buffer.
        w.Writers.Unlock()
    }
}
```

No código 4.5 é mostrado a implementação do escritor, que começa um loop infinito em que ele pede acesso ao recurso por meio da função `Lock` do *mutex*. Ao obter o acesso, ele realiza a escrita. Por fim, ele libera o acesso ao recurso chamando a função `Unlock`.

Código 4.6 – Código do leitor para implementação básica

```

package main

import (
    "sync"
)

type Reader struct {
    ID      int
    Count   chan int
    Writers *sync.Mutex
}

func (r *Reader) start() {
    for {
        count := <-r.Count
        if count == 0 {
            r.Writers.Lock()
        }
        r.Count <- count + 1

        // realiza leitura

        count = <-r.Count
        if count == 1 {
            r.Writers.Unlock()
        }
        r.Count <- count - 1
    }
}

```

No código 4.6, tem-se o código dos leitores. Há uma diferença em relação ao código 4.5, pois deseja-se que todos os leitores possam ler o dado concorrentemente, e assim precisa-se fazer com que apenas o primeiro leitor chame a função `Lock` do *mutex*. Para isso, utiliza-se o canal `Count` para armazenar a quantidade de leitores em um determinado momento. Esse canal é iniciado com tamanho 1 e com o valor 0. Assim, todo leitor que inicia deve consumir o valor do canal, acrescentar em uma unidade esse valor, e escrevê-lo no canal. Após a leitura, ele deve novamente pegar a quantidade de leitores por meio do canal `Count` e verificar se ele é o último leitor acessando o recurso. Se sim, ele deve liberar o recurso para os escritores. Por fim, ele decrementa o valor atual do buffer `Count`.

A opção por uso de canal para guardar a quantidade de leitores num determinado momento é que, neste cenário, existe um dado a ser compartilhado entre as *goroutines*, que é a quantidade de leitores. Assim, a solução convencional seria criar seções críticas para acessar o contador, porém as características bloqueantes dos canais já garantem a

exclusão mútua na leitura do valor do canal.

Código 4.7 – Código da função principal para implementação básica

```
package main

import "sync"

func main() {
    numReaders := 10 // valor qualquer
    numWriters := 10 // valor qualquer

    readers := make(chan int, 1)
    readers <- 0
    writers := &sync.Mutex{}

    for i := 0; i < numWriters; i++ {
        writer := &Writer{ID: i + 1, Writers: writers}
        go writer.start()
    }
    for i := 0; i < numReaders; i++ {
        reader := &Reader{ID: i + 1, Count: readers, Writers: writers}
        go reader.start()
    }
    select {}
}
```

O código 4.7 da função principal faz as inicializações do problema. Essa função principal fica também responsável por criar o *mutex* e o canal que serão compartilhados entre as *goroutines* dos leitores e dos escritores.

Seria possível também utilizar um canal para fazer o bloqueio de acesso dos escritores e dos leitores. Entretanto, dado que canais utilizam implementações de semáforos ou *mutexes*, dependendo da arquitetura do sistema, semelhantes aos utilizados no problema, não há um ganho específico em seu uso nesse caso. Não há informação a ser transmitida, o uso do canal se daria apenas pela sua característica bloqueante.

4.1.2.2 Leitores e escritores com prioridade para escrita

Na seção 4.1.2.1, a lógica para implementação básica obriga os escritores a esperarem todos os leitores. Os leitores, podem começar a ler sem retenção. A solução básica então não resolve problemas onde não podem haver *starvation* de escritores. Um exemplo é um programa que registra transações financeiras e para completar o registro deve se escrever num arquivo. Esse arquivo deve sempre ser priorizado para escrita, pois não se pode perder as transações.

O problema é resolvido de maneira semelhante ao anterior. São feitas alterações nas condições de mudança do fluxo do escritor e do fluxo do leitor, de maneira a priorizar quando houver um fluxo tentando escrever no recurso. O fluxo proposto é:

- Caso um processo **leitor** requisite o acesso ao recurso, ele irá:
 - Ler o recurso, caso não tenha nenhum outro processo acessando o recurso;
 - Ler o recurso, caso tenha um ou mais leitores utilizando e não tenha um escritor aguardando;
 - Aguardar a execução de todos os escritores, caso tenha escritores aguardando ou utilizando o recurso.
- Caso um processo **escritor** requisite o acesso ao recurso, ele irá:
 - Escrever no recurso, caso não tenha nenhum outro processo utilizando o recurso;
 - Aguardar a liberação do recurso, caso tenha um escritor acessando o recurso;
 - Aguardar o fim dos leitores sem permitir que novos sejam iniciados, caso tenha pelo menos um leitor acessando o recurso.

Código 4.8 – Código do escritor com prioridade para escrita

```

package main

import (
    "sync"
)

type Writer struct {
    ID      int
    Writers *sync.Mutex
    Readers *sync.Mutex
    Count   chan int
}

func (w *Writer) start() {
    for {
        count := <-w.Count
        if count == 0 {
            w.Readers.Lock()
        }
        w.Count <- count + 1
        w.Writers.Lock()
        // realiza a escrita
        w.Writers.Unlock()
        count = <-w.Count
        if count == 1 {
            w.Readers.Unlock()
        }
        w.Count <- count - 1
    }
}

```

O código 4.8 do escritor com prioridade fica semelhante ao código 4.5, porém com um adição importante: agora, o primeiro escritor sempre deve pedir um novo mutex, chamado de **Readers**. Utiliza-se um contador dos escritores aguardando, semelhante ao contador de leitores no código 4.6. Ao conseguir adquirir o *mutex Readers*, ele pede acesso ao recurso com os outros escritores, usando o *mutex Writers*. Assim, temos que diversos escritores irão competir entre si apenas por meio do *mutex Writers*, de acesso ao recurso, enquanto os leitores aguardarão todos os escritores finalizarem e assim obter o *mutex Reader* antes de conseguir ler do recurso.

Código 4.9 – Código do leitor com prioridade para escrita

```

package main

import (
    "sync"
)

type Reader struct {
    ID      int
    Readers *sync.Mutex
    Writers *sync.Mutex
    Count   chan int
}

func (r *Reader) start(stream []byte, size int) {
    for {
        r.Readers.Lock()
        count := <-r.Count
        if count == 0 {
            r.Writers.Lock()
        }
        r.Count <- count + 1
        r.Readers.Unlock()
        // realiza a leitura
        count = <-r.Count
        if count == 1 {
            r.Writers.Unlock()
        }
        r.Count <- count - 1
    }
}

```

O código 4.9 do leitor sem prioridade é ainda mais semelhante ao código 4.6, ainda com uma relevante diferença: antes da lógica de leitura, é executado o comando de `Lock` no *mutex* `Readers`. Assim, caso os escritores estejam escrevendo, todo leitor irá aguardar a liberação deste *mutex*. Quando um leitor for liberado, ele pedirá o mutex do recurso, chamado de `Writers`, impedindo escritores que chegaram depois da obtenção do *mutex* `Readers`. Ele utiliza a mesma lógica anterior do código 4.6 para permitir diversos readers lendo simultaneamente. Por fim, antes de iniciar a leitura do recurso, ele libera o mutex `Readers`. Se um escritor conseguir adquiri-lo, nenhum outro leitor conseguirá ler o recurso. Caso contrário, mais leitores poderão realizar a leitura.

Código 4.10 – Código da função principal com prioridade para escrita

```

package main

import (
    "sync"
)

func main() {
    numReaders := 10 // valor qualquer
    numWriters := 10 // valor qualquer

    readers := &sync.Mutex{}
    writers := &sync.Mutex{}
    readerCount := make(chan int, 1); readerCount <- 0
    writerCount := make(chan int, 1); writerCount <- 0

    for i := 0; i < numWriters; i++ {
        writer := &Writer{ID: i + 1, Readers: readers, Writers: writers,
            Count: writerCount}
        go writer.start()
    }
    for i := 0; i < numReaders; i++ {
        reader := &Reader{ID: i + 1, Readers: readers, Writers: writers,
            Count: readerCount}
        go reader.start()
    }
    select {}
}

```

O código 4.10 da função principal realiza o mesmo que o código 4.7 e além disso também cria o canal de contagem de escritores e o *mutex Readers*, utilizados para garantir prioridade aos escritores.

4.1.2.3 Leitores e escritores sem starvation

Por fim, propomos uma solução para o problema dos leitores e escritores sem nenhuma prioridade, ou sem criar *starvation* a nenhum dos dois fluxos. Assim, antes de um leitor iniciar a leitura enquanto outro leitor está consumindo, ele deve verificar se um escritor está querendo acessar o recurso. O mesmo deve ser feito para o escritor, caso algum escritor já esteja acessando o recurso e tenha leitores aguardando o próximo escritor deve aguardar a execução dos leitores. Deste maneira, há sempre uma alternância entre leitores e escritores que acessam o recurso.

A proposta de lógica dos fluxos é:

- Caso uma *thread* **leitor** requisiite o acesso ao recurso, ele irá:
 - Ler o recurso, caso não tenha nenhuma outra *thread* acessando o recurso.
 - Ler o recurso, caso tenha um ou mais leitor acessando-o e não tenha escritores aguardando.
 - Aguardar a liberação do recurso, caso um escritor esteja acessando o recurso.
 - Aguardar o início de um escritor, caso tenha algum escritor aguardando acesso ao recurso.
- Caso uma *thread* **escritor** requisiite o acesso ao recurso, ele irá:
 - Escrever no recurso, caso não tenha nenhum outra *thread* acessando o recurso
 - Aguardar a liberação do recurso, caso um escritor esteja acessando o recurso.
 - Aguardar a liberação do recurso e avisar os leitores que há um escritor esperando, caso haja leitores acessando o recurso.

Felizmente, para este tipo de cenário, a linguagem Go já providencia um tipo de *mutex* específico, chamado de **RWMutex**, ou um *mutex* de escrita e leitura (PIKE et al.,). Os leitores podem utilizar um tipo próprio de função para obtenção do *mutex* chamada de **RLock** e os escritores devem acessá-lo a partir da função **Lock**. Há também uma função própria de liberação do *lock* para os leitores, chamada **RUnlock**, e a função de liberação do *lock* para o escritor é chamada **Unlock**.

A garantia de não ter *starvation* é obtida por meio das regras de acesso do RWMutex, de acordo com a função utilizado. Caso exista um leitor tentando obter o *mutex* por meio da função **RLock**, é realizada a verificação de que não existe escritor acessando ou aguardando o recurso, impedindo assim o acesso para leitura caso exista ao menos um escritor esperando. Na liberação do RWMutex utilizando a função **Unlock** no caso dos escritores, antes de liberar um *mutex* de sincronia dos escritores, este escritor que estava acessando o recurso libera os leitores para que eles possam iniciar leitura. Assim, há uma garantia de alternância entre as *threads* acessando o recurso.

Este tipo de *mutex* é mais custoso que os convencionais. A lógica que seria implementado pelo desenvolvedor é na realidade abstraída pela linguagem. Assim, é recomendado o uso deste tipo num cenário onde há mais leitura do que escrita e os fluxos estão sobre contenção, ou seja, existe um grande acesso de leitura e de escrita ao recurso, consequentemente constantemente trancando e destrancando o *mutex* (DONOVAN; KERNIGHAN, 2015).

Código 4.11 – Código do escritor para implementação sem *starvation*

```

package writer
import (
    "sync"
)
type Writer struct {
    ID      int
    Mutex   *sync.RWMutex
}
func (w *Writer) start() {
    for {
        w.Mutex.Lock()
        // realiza a escrita
        w.Mutex.Unlock()
    }
}

```

No código 4.11, referente à função dos escritores, toda a lógica de contadores é substituído apenas pela chamada de `Lock` no *mutex* compartilhado entre as *threads*.

Código 4.12 – Código do leitor para implementação sem *starvation*

```

package reader
import (
    "sync"
)
type Reader struct {
    ID      int
    Mutex   *sync.RWMutex
}
func (r *Reader) start() {
    for {
        r.Mutex.RLock()
        // realiza a leitura
        r.Mutex.RUnlock()
    }
}

```

No código 4.12 referente à função dos leitores também toda a lógica dos contadores e de múltiplos *mutexes* é substituída pelas chamadas das funções `RLock` e `RUnlock`. Estas funções permitem um uso síncrono da seção crítica seguindo as premissas do problema: diversos leitores podem acessar simultaneamente, porém apenas um escritor pode escrever por vez, e nenhum leitor pode acessar o recurso enquanto estiver ocorrendo a escrita. A lógica fica semelhante ao código 4.6, porém utilizando outra estrutura de dado e sem a necessidade do contador.

Código 4.13 – Código da função principal para implementação sem *starvation*

```

package main
import (
    "sync"
)
func main() {
    numReaders := 10 // valor qualquer
    numWriters := 10 // valor qualquer
    mutex := &sync.RWMutex{}
    for i := 0; i < numWriters; i++ {
        writer := Writer{ID: i + 1, Mutex: mutex}
        go writer.start()
    }
    for i := 0; i < numReaders; i++ {
        reader := &Reader{ID: i + 1, Mutex: mutex}
        go reader.start()
    }
    select {}
}

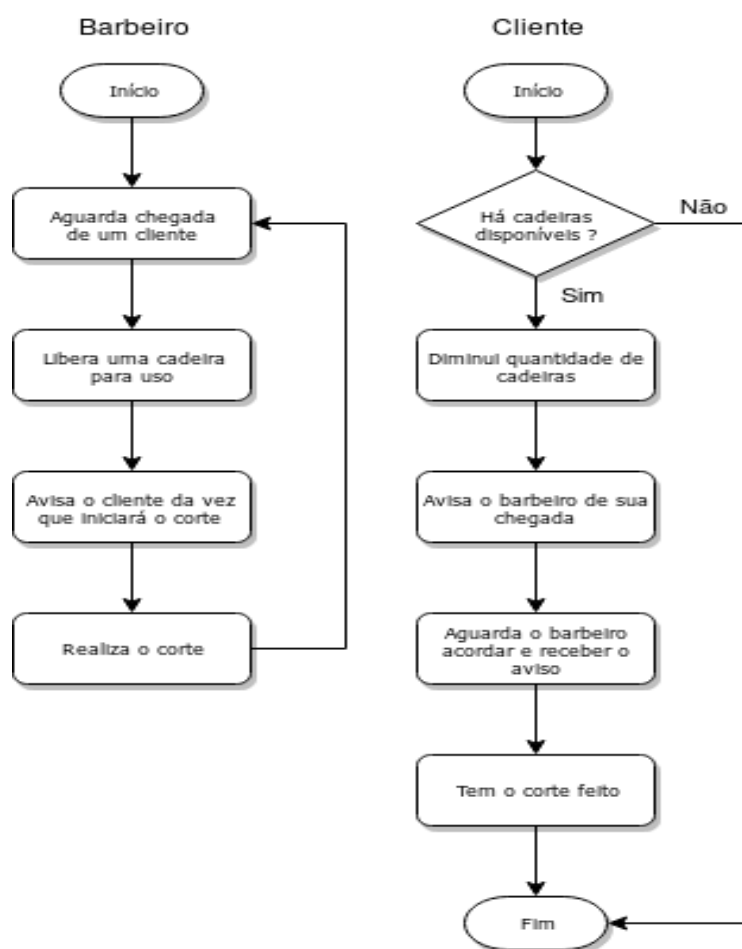
```

O código 4.13 da função principal realiza menos inicializações comparado ao código 4.7 e ao código 4.10, seguindo a lógica dos outros dois para início do programa.

4.1.3 Barbeiro dorminhoco

O problema do barbeiro dorminhoco é um outro tipo de problema clássico de concorrência. Neste problema, existem dois tipos de processos: barbeiro e clientes. O barbeiro deve dormir enquanto não houver nenhum cliente para ser atendido. Na chegada do cliente, o barbeiro é acordado e começa o corte de cabelo. Se novos clientes chegarem enquanto o barbeiro estiver realizando um corte, eles devem ser enfileirados até um número máximo de “cadeiras” que o salão providencia.

O fluxo esperado de cada *thread* é indicado na figura 11, mostrando o fluxo de um barbeiro e de um cliente. Observe que o barbeiro opera indefinidamente, sem uma condição final para a *thread*.

Figura 11 – Fluxograma das *threads* do barbeiro dorminhoco

Código 4.14 – Código do barbeiro

```

package main

type Barber struct {
    ID int
}

func (b *Barber) start(barberReady, costumerReady, seatsCh chan int) {
    for {
        // aguarda cliente
        costumerID := <-costumerReady // aguarda um cliente ficar pronto
        seatsCh <- <-seatsCh + 1 // libera uma cadeira
        barberReady <- b.ID // avisa ao cliente que iniciar o corte
        // realiza o corte
    }
}
  
```

No código 4.14, há a declaração do tipo **Barber**, a estrutura de dados responsável pela lógica do barbeiro. Ele possui um ID, e um canal que é utilizado para avisar aos clientes que esse barbeiro se encontra pronto para iniciar o corte. Seu fluxo então é:

- Aguarda a chegada de um cliente;
- Ao chegar o cliente, avisa-o de que está pronto para cortar;
- Corta o cabelo, e reinicia o fluxo.

Numa implementação convencional utilizando *mutexes* ou semáforos, normalmente o problema é atacado criando-se seções críticas em cada *thread* para modificar a quantidade de assentos livres. Em Go, é possível modelar como uma troca de mensagens entre os barbeiros e os clientes, e utilizando o recurso de canais não é necessário a criação destas seções críticas. Utiliza-se um canal para comunicação do cliente e do barbeiro, indicando que há um cliente pronto para o corte, e utiliza-se um canal para comunicação de que o barbeiro iniciará o corte no cliente. Por fim, utiliza-se um canal com *buffer* de tamanho 1, para guardar a quantidade de assentos livres num determinado instante. Usando estes mecanismos, seria possível inclusive resolver a variação do problema onde existem diversos barbeiros.

Código 4.15 – Código do cliente

```
package main

type Costumer struct {
    ID int
}

func (c Costumer) start(barberReady, costumerReady, seatsCh chan int) {
    seats := <-seatsCh // pega n mero atual de cadeiras dispon veis
    if seats > 0 {
        seatsCh <- seats - 1 // diminui cadeiras dispon veis
        costumerReady <- c.ID // avisa ao barbeiro sua chegada
        barberID := <-barberReady // aguarda o barbeiro acordar e avisa-
            lo do inicio
            // tem o corte feito
    } else {
        seatsCh <- seats
        // sem lugar para aguardar, desiste do corte
    }
}
```

No código 4.15, o fluxo esperado é:

- O cliente ao chegar confere a quantidade de cadeiras. Se tiver algum lugar, fica aguardando. Senão, vai embora;

- Aguarda sua vez de ser atendido;
- Quando chegar sua vez, avisa ao barbeiro e inicia o corte.

Novamente, se faz uso de canais para dois cenários: um para comunicar as mudanças de estado do cliente; e outro canal para transmitir o número atual de cadeiras livres. Neste caso, basta decrementar o valor armazenado no canal, caso um cliente chegue; ou manter o mesmo valor, caso não haja cadeira livre.

Código 4.16 – Código da função principal do barbeiro dorminhoco

```
package main

const NUMBER_OF_COSTUMERS = 1000
const NUMBER_OF_SEATS = 15

import (
    "time"
)

func main() {
    barberReady := make(chan int)
    costumerReady := make(chan int)
    seatsCh := make(chan int, 1)
    seatsCh <- NUMBER_OF_SEATS
    barber := &Barber{ID: 1}
    go barber.start(barberReady, costumerReady, seatsCh)
    for i := 0; i < NUMBER_OF_COSTUMERS; i++ {
        costumer := Costumer{ID:i + 1}
        go costumer.start(barberReady, costumerReady, seatsCh)
    }
    select {}
}
```

Por fim, temos a função principal no código 4.16. Nesta função estão as inicializações dos clientes, do barbeiro, dos canais utilizados na comunicação das *goroutines* e da quantidade de assentos. É feito também a inicialização do canal referente às cadeiras livres, com *buffer* de tamanho 1, e valor inicial igual ao número total de cadeiras, no caso definido como 15.

4.2 PROGRAMAÇÃO DISTRIBUÍDA

Até aqui, foi apresentado como usar as primitivas de Go para resolver problemas clássicos de concorrência. Nesta seção, serão apresentados exemplos de padrões de programação distribuída, mostrando as funcionalidades das bibliotecas básicas de Go para interação entre processos que podem executar em máquinas distintas.

4.2.1 Aplicação cliente/servidor

Em uma aplicação cliente/servidor temos dois tipos de processos: servidores, responsáveis por prover algum tipo de recurso ou serviço; e clientes que requisitam o recurso ou serviço provido. Neste caso, recurso é qualquer tipo de processamento a ser executado ou dado a ser transmitido, como por exemplo, páginas *HTML*, consultas a banco de dados, chamadas de procedimento remotos (RPC), entre outros. Esses processos podem estar em computadores diferentes, realizando a troca de mensagens a partir de uma rede de comunicação.

Neste tipo de modelo de aplicação, é comum a comunicação ser realizada a partir de *sockets* oferecidos pelo sistema operacional, usando os serviços da camada de transporte da rede por meio dos protocolos UDP ou TCP.

Código 4.17 – Exemplo de servidor

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "strings"
)

func main() {
    ln, _ := net.Listen("tcp", ":3000")
    for {
        conn, err := ln.Accept()
        if err != nil {
            fmt.Println("Can't listen to socket")
            break
        }
        go func() {
            message, _ := bufio.NewReader(conn).ReadString('\n')

            fmt.Print("Message received:", string(message))

            newMessage := strings.ToUpper(message)

            conn.Write([]byte(newMessage + "\n"))
        }()
    }
}
```

No código 4.17, temos a função principal do programa servidor. Ele reserva um socket na porta 3000, com endereço de IP qualquer, num *loop* infinito que aceita as conexões

que chegam e, caso não ocorra nenhum erro, lê os *bytes* enviados por este *socket*. Depois, transforma os *bytes* recebidos numa *string* e envia o valor desse texto em caixa alta, convertido novamente para *bytes*, de volta para quem enviou o pacote. Como está sendo utilizado o tipo de socket TCP, pode-se escrever uma resposta.

Código 4.18 – Exemplo de cliente

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
)

func main() {
    conn, err := net.Dial("tcp", "127.0.0.1:3000")
    if err != nil {
        fmt.Println("Server is down")
        os.Exit(1)
    }
    reader := bufio.NewReader(os.Stdin)
    fmt.Print("Text to send: ")
    text, _ := reader.ReadString('\n')

    fmt.Fprintf(conn, text+"\n")

    message, _ := bufio.NewReader(conn).ReadString('\n')
    fmt.Print("Message from server: " + message)
}
```

Já no código 4.18, tem-se a função do programa cliente. Ele funciona de maneira mais simples. Utilizando novamente a biblioteca **net**, é feito a requisição de se conectar por um *socket* TCP na porta 3000 no endereço local (ou seja, na mesma máquina). Em caso de erro, termina o programa. Após corretamente conseguir conectar-se ao socket, o fluxo aguarda o *input* de texto do usuário que será enviado para o servidor. Após a conclusão do *input*, é feito o envio do texto convertido para uma *stream* de *bytes*, e por fim é feito a leitura da resposta, sua conversão para *string* e a impressão.

A biblioteca padrão **net** já possui uma solução completa para essa comunicação com *sockets*, e o uso das *goroutines* para responder os sockets concorrentemente permite uma solução simples para a criação de fluxos triviais de sistemas distribuídos.

4.2.2 Chat

Chat é um tipo de sistema de troca de mensagens instantâneas, onde vários usuários trocam mensagens por meio de uma interface de entrada e saída de dados. As mensagens podem ser centralizadas num servidor, e esse fica responsável por repassá-las para os outros clientes; ou as mensagens são passadas diretamente entre os clientes.

O chat funciona semelhantemente ao exemplo apresentado de servidor e cliente: um cliente irá mandar uma mensagem pro servidor, e este reproduzirá para todos os clientes que estejam conectados ao chat no momento (diferentemente do exemplo anterior, que é só uma troca de mensagem entre o cliente e o servidor). Para isso, é necessário salvar quais são os clientes atualmente conectados, e também removê-los caso algum cliente saia.

Código 4.19 – Código da função principal do servidor do chat

```

package main
import (
    "bufio"
    "fmt"
    "log"
    "math/rand"
    "net"
    "strconv"
    "sync"
    "time"
)

func main() {
    server, err := net.Listen("tcp", ":3000")
    if err != nil {
        panic(err)
    }
    clients := &sync.Map{}
    dead := make(chan string)
    messages := make(chan string)
    rand.Seed(time.Now().UTC().UnixNano())
    go receivePeople(server, clients, messages, dead)
    for {
        select {
        case message := <-messages:
            clients.Range(func(name, conn interface{}) bool {
                go func(conn net.Conn) {
                    _, err := conn.Write([]byte(message))
                    if err != nil {
                        dead <- name.(string)
                    }
                }(conn.(net.Conn))
                return true
            })
            log.Printf("New message: \"%s\"\n", message[:len(message)-2])
        case name := <-dead:
            go func() {
                log.Printf("Client %s disconnected\n", name)
                clients.Delete(name)
                messages <- fmt.Sprintf("O usuario %s saiu do chat.\n", name)
            }()
        }
    }
}

```

O código 4.19 da função principal é responsável por inicializar o servidor, que escuta uma porta qualquer (no exemplo, a porta 3000). É responsável também por criar o mapa que guarda quais usuários estão conectados. Este mapa é uma estrutura especial disponibilizado pela biblioteca *sync*, que permite um uso síncrono de uma estrutura de mapa entre *goroutines*. Os canais a serem utilizadas para comunicar a saída de um usuário e o envio de novas mensagens também são criados na função principal. A função `receivePeople` fica responsável por lidar com a chegada de novos usuários, e é chamada pela *main* criando uma nova *goroutine*, para permitir o fluxo principal da *main* concorrentemente com a chegada de novos clientes.

O servidor então fica num *loop* entre duas possibilidades: receber alguma mensagem no canal de mensagens, ou receber um aviso de usuário cuja conexão foi encerrada. No caso de uma nova mensagem, ele itera pelo mapa dos usuários que estão no chat e escreve a mensagem no socket para cada um deles. Já quando é notificado de uma conexão morta, ele deleta o cliente do mapa.

Código 4.20 – Código da função de recebimento de novos usuarios do chat

```

func receivePeople(server net.Listener, clients *sync.Map, messages chan
string, dead chan string) {
    for {
        conn, err := server.Accept()
        if err != nil {
            panic(err)
        }
        go func(conn net.Conn) {
            reader := bufio.NewReader(conn)
            _, err = conn.Write([]byte("Welcome to the chat! Please,
            type your name!\n"))
            if err != nil {
                return
            }
            name, err := reader.ReadString('\n')
            if err != nil {
                return
            }
            name = name[:len(name)-2]
            if _, ok := clients.Load(name); ok {
                for {
                    temp := name + strconv.Itoa(int(rand.Int31n(10e07)))
                    if _, ok = clients.Load(temp); ok {
                        continue
                    }
                    name = temp
                    break
                }
            }
            clients.Store(name, conn)
            log.Printf("New client: %s!\n", name)

            messages <- fmt.Sprintf("0 usuario %s entrou do chat.\n",
            name)
            for {
                message, err := reader.ReadString('\n')
                if err != nil {
                    break
                }
                messages <- fmt.Sprintf("[%s]: %s", name, message)
            }
            dead <- name
        }(conn)
    }
}

```

Por fim, o código 4.20 da função de novos usuários implementa um *loop* que aceita novas conexões ao *socket* sendo escutado. Para cada nova conexão aceita, o servidor pede ao cliente um nome para ser identificado. Caso o nome já exista, é gerado um sufixo aleatório até que o nome seguido do sufixo não exista no mapa. Por fim, a função manda uma nova mensagem pro canal de mensagens, que avisa a chegada de um novo usuário para outros clientes. Por fim, ele inicializa a função do novo usuário, que irá ficar em *loop* ouvindo aquela conexão, até que ocorra algum erro. Quando ocorre um erro, o usuário é adicionado no canal de conexões derrubadas. Todo esse tratamento, após ser aceito a nova conexão, é feito assincronamente, para permitir o tratamento de múltiplos clientes chegando simultaneamente.

Neste exemplo de chat, é possível ver um uso completo das primitivas do Go. Tanto o uso da biblioteca *sync* e primitivas poderosas como a de um mapa de acesso síncrono, como a primitiva de seleção para decidir qual será a próxima ação do servidor, e o uso de *goroutines* para permitir facilmente que todos esses fluxos sejam concorrentes: o código não irá travar na espera do *input* do usuário sem enviar mensagem para os usuários do chat.

5 CORROTINAS

Corrotina é uma proposta de abstração de controle, introduzida no início dos anos 1960. É atribuído a Conway, que descreveu corrotinas como "subrotinas que agem como o programa mestre" (MOURA; IERUSALIMSKY, 2009). As características fundamentais definidas por Christopher Marlin (MARLIN, 1980) das corrotinas são:

- Os dados locais de uma corrotina persistem entre sucessivas chamadas;
- A execução de uma corrotina é suspensa quando o controle a deixa, e só retorna no mesmo local que foi deixado. Isto é, no momento que é o fluxo deixa a execução de uma corrotina, ela só deverá retornar, para a instrução seguinte da parada, quando for retomada por outra corrotina.

Corrotinas são um tipo de controle de fluxo diferentes de processos e *threads* devido a sua característica cooperativa: um fluxo qualquer pausa sua execução e permite um outro fluxo executar em seu lugar. Assim, muitos problemas de condição de corrida podem ser evitados, ao mesmo tempo que permite concorrência mas não necessariamente permite paralelismo.

O tipo específico de corrotinas que será abordado são as corrotinas assimétricas. Corrotinas simétricas e corrotinas assimétricas se distinguem a partir das funções de controle de mudança e fluxo. Na corrotina simétrica existe apenas uma única operação que permitem cada uma transferir o controle a outra. Na corrotina assimétrica, existem dois tipos de operação: a que transfere o fluxo a uma outra corrotina e uma que suspende a execução da corrotina atual. Quando a corrotina suspende sua execução, ela retorna o controle para a corrotina que a invocou. Dessa maneira, há uma espécie de hierarquia, no caso assimétrico, entre a corrotina que foi invocada e a que invocou. Enquanto isso, na simétrica todas operam no mesmo nível hierárquico.

Lua disponibiliza corrotinas assimétricas completas que, como demonstrado em (MOURA; IERUSALIMSKY, 2009), possibilitam criar *one shot continuations*, ou continuações de um turno, subcontinuações, e corrotinas simétricas a partir destas corrotinas assimétricas. Assim, foi criado uma proposta de corrotina assimétrica em Go, modelada de forma semelhante ao funcionamento das corrotinas em Lua.

5.1 IMPLEMENTAÇÃO EM GO

Código 5.1 – Código de definição e criação de corrotinas em Go

```
package coroutines

type Coroutine struct {
    base          func(...interface{}) []interface{}
    yield         chan interface{}
    resume        chan interface{}
    dead, started bool
}

func Create(base func(*Coroutine, ...interface{}) []interface{}) *
    Coroutine {
    coro := &Coroutine{
        yield:  make(chan interface{}),
        resume: make(chan interface{}),
        dead:   false,
        started: false,
    }
    coro.base = func(args ...interface{}) []interface{} {
        rets := base(coro, args...)
        coro.dead = true
        return coro.Yield(rets...)
    }
    return coro
}
```

O código 5.1 demonstra a criação de uma estrutura de corrotina. A estrutura guarda qual foi a função utilizada para construção (chamada de **base**), dois canais utilizados para transferência de dados, **yield** e **resume** e por fim duas flags, que indicam o estado da corrotina.

Na criação da corrotina, para que fosse permitido ter a visibilidade da variável sendo utilizada (no caso, a variável de retorno **coro**), foi separado a criação da estrutura e a atribuição da função passada como argumento. Para que a função pudesse aceitar parâmetros indefinidos, e ainda retornar uma lista qualquer de parâmetros, foi utilizado o recurso de um argumento variáveis do tipo de uma interface vazia. O primeiro argumento, porém, deve ser a própria corrotina. Isso é necessário para fazer chamadas de **Yield** dentro da corrotina, já que é necessário uma referência para essa função, que não está disponível no momento da chamada da função **Create**.

Código 5.2 – Continuação do código de corrotinas: *resume* e *yield* das corrotinas

```

func (c *Coroutine) Resume(args ...interface{}) []interface{} {
    if c.dead {
        panic("Cannot resume a dead coroutine")
    }
    if !c.started {
        c.started = true
        go c.base(args...)
    } else {
        for _, value := range args {
            c.resume <- value
        }
        close(c.resume)
    }
    list := []interface{}{}
    for yieldedValue := range c.yield {
        list = append(list, yieldedValue)
    }
    c.yield = make(chan interface{})
    return list
}

func (c *Coroutine) Yield(rets ...interface{}) (list []interface{}) {
    for _, value := range rets {
        c.yield <- value
    }
    close(c.yield)
    if len(rets) > 0 {
        list = []interface{}{}
        for newArg := range c.resume {
            list = append(list, newArg)
        }
    }
    c.resume = make(chan interface{})
    return list
}

```

No código 5.2, é possível ver as implementações das funções *resume* e *yield*, finalizando as três funções básicas implementadas em Lua. A função *resume* é utilizada tanto para iniciar, quanto para reativar as corrotinas, e não tem efeito quando chamada após o encerramento da corrotina. Na sua primeira execução, a função *resume* deve sempre chamar a função da corrotina por meio de uma *goroutine*, para impedir *deadlocks* de espera de canais. Ao final de toda chamada do *resume*, coleta-se os dados produzidos pela corrotina, que pode ser uma chamada para a função *yield* vinda de dentro da função base, ou a chamada de *yield* feita após a finalização da função base da corrotina. Para

utilizar o recurso de **range** em canais, é necessário um fechamento dos canais, e por não haver uma maneira segura de saber se um canal encontra-se fechado, foram utilizados dois canais, em que cada uma das rotinas que executam a leitura fica responsável pela recriação desse canal, e as rotinas que escrevem ficam responsáveis pelo fechamento do canal.

5.2 EXEMPLO DE USO DE CORROTINAS EM GO

Para avaliar a implementação proposta de corrotinas em Go, reproduzimos a solução para o problema de percorrer uma árvore binária por meio de corrotinas, apresentado em (MOURA; IERUSALIMSKY, 2009).

O código 5.3 mostra a implementação em Go para percorrer simultaneamente duas árvores binárias. A sequência de caminhamento nas árvores deve ser a seguinte: os dois índices são obtidos inicialmente das duas árvores. O de menor valor é impresso, e se os valores forem iguais, escolhe-se o elemento da primeira árvore. Quando uma árvore termina sua impressão, ela deixa de ser percorrida. O programa termina quando o caminhamento nas duas árvores é encerrado.

Apesar de algumas idiossincrasias do Go, como de tratar variáveis que podem ser de qualquer valor, por meio de *interfaces* vazias, a lógica da solução de referência em Lua não é afetada. Primeiro, é realizado a criação das árvores iniciais. Depois, é definido a função base da corrotina, chamada de **find**, que apenas faz o acesso no nó da esquerda, retorna o valor por meio do *yield*, e depois acessa o nó da direita. Caso o nó atual seja nulo, a função apenas retorna nulo.

Após isso, é feito a criação das duas corrotinas: uma para percorrer a árvore A, e outra para percorrer a árvore B. Nota-se que aqui ainda não iniciou a execução da rotina. É coletado então os primeiros valores das árvores. Como o retorno é uma lista, o loop funciona enquanto a lista retornada tiver um tamanho maior que 0. Após isso, ele apenas checa se deve imprimir o valor da árvore A ou da árvore B, e chama novamente a corrotina referente ao nó da árvore impresso.

Código 5.3 – Exemplo de corrotinas: percorrendo árvores binárias

```

package main

import (
    "fmt"
    "tcc-coroutines/coroutines"

    "golang.org/x/tour/tree"
)

func main() {
    var find func(*coroutines.Coroutine, ...interface{}) []interface{}

    A := tree.New(2)
    B := tree.New(3)

    find = func(c *coroutines.Coroutine, args ...interface{}) []
        interface{} {
            t := (args[0]).(*tree.Tree)
            if t != nil {
                find(c, t.Left)
                c.Yield(t.Value)
                find(c, t.Right)
            }
            return nil
        }

    stepTree1 := coroutines.Create(find)
    stepTree2 := coroutines.Create(find)
    v1 := stepTree1.Resume(A)
    v2 := stepTree2.Resume(B)
    for len(v1) > 0 || len(v2) > 0 {
        if len(v1) > 0 && (len(v2) == 0 || v1[0].(int) < v2[0].(int)) {
            fmt.Printf("%v, ", v1[0])
            v1 = stepTree1.Resume()
        } else {
            fmt.Printf("%v, ", v2[0])
            v2 = stepTree2.Resume()
        }
    }
    fmt.Printf("\n")
}

```

Comparando o código em Lua apresentado em (MOURA; IERUSALIMSKY, 2009) e o código em Go utilizando o recurso de corrotina criado, é possível notar uma maior simplicidade no tratamento da passagem de variáveis de dentro para fora da corrotina no

código em Lua. Devido à característica do Go de código estaticamente tipado e por não possuir um recurso de tipo genérico, é necessário essa manipulação de variáveis do tipo **interface** no momento de compilação, permitindo que a variável possua qualquer tipo e valor em tempo de execução.

Em relação às limitações, a solução proposta em Go implementa completamente o recurso de corrotinas assimétricas, que podem receber um número qualquer de argumentos entre as corrotinas e código principal. Então, não há nenhuma limitação vinda da implementação em Go.

6 AVALIAÇÃO DE DESEMPENHO DE GO

Nesta seção, será analisado o desempenho de *goroutines*, o principal recurso de concorrência da linguagem Go. Para isso, será utilizado o problema clássico de multiplicação de matrizes. O principal aspecto analisado será o tempo de execução do programas, sendo posteriormente comparado com uma solução do mesmo problema na linguagem C++.

6.1 MULTIPLICAÇÃO DE MATRIZES

No problema de multiplicação de matrizes não há necessidade de sincronização entre os diversos fluxos de execução, dado que há independência de cálculo de cada campo da matriz de saída. Assim, o que será analisado é o custo de administração das *goroutines* pelo ambiente do Go.

Considere uma matriz qualquer A , bidimensional, de tamanho N por M . Considere também uma matriz B qualquer de tamanho M por N' , e considere a matriz resultado $C = A * B$, com dimensões N por N' .

Para cada par i, j , onde $1 \leq i \leq N$ e $1 \leq j \leq N'$, sendo $C_{i,j}$ um elemento da matriz de saída, temos que o valor desse elemento é igual a:

$$C_{i,j} = \sum_{k=1}^M (A_{i,k} * B_{k,j}) \quad (6.1)$$

Em que $A_{i,k}$ é um elemento da matriz A e $B_{k,j}$ é um elemento da matriz B .

Assim, é possível observar que cada elemento da matriz final C depende exclusivamente dos valores lidos de A e de B . Dado que as matrizes A e B não são alteradas a partir do início da multiplicação, o cálculo de cada campo é independente da execução do outro campo.

6.2 CÓDIGO EM GO

Foi escolhido um algoritmo ingênuo de implementação de matrizes, em que é criado uma *goroutine* para cada índice da matriz final. A ideia é explorar um problema com alto paralelismo de dados, que ainda seja simples sua implementação. Posteriormente, é feito também uma implementação com um algoritmo mais clássico de separação de blocos.

Além disso, a ideia é também analisar como a linguagem se comporta com a criação de um alto número de *goroutines*, e se há uma degradação de desempenho. Porém, este tipo de algoritmo é comumente mais lento do que outros algoritmos. Utilizando *kernel threads* não seria possível sua execução para matrizes grandes, dado ao grande custo de memória que estes tipos de *threads* possuem.

Código 6.1 – Código da estrutura *Matrix* em Go

```

package main

import (
    "sync"
)

type MatrixInt [][]int
func (c MatrixInt) Step(a, b MatrixInt, i, j int, wg *sync.WaitGroup) {
    defer wg.Done()
    for j := 0; j < len(a[0]); j++ {
        c[i][j] += a[i][0] * b[0][j]
    }
}
func (c MatrixInt) Multiply(a, b MatrixInt) {
    wg := &sync.WaitGroup{}
    for i := 0; i < len(a); i++ {
        for k := 0; k < len(a[0]); k++ {
            wg.Add(1)
            go c.Step(a, b, i, j, wg)
        }
    }
    wg.Wait()
}

```

No código 6.1, foi criado o tipo `MatrixInt`, que é apenas um *alias* para um *slice* de `int`, como visto na seção 3.1.2. Foram definidos então dois métodos para esse tipo de dado, utilizados para cálculo da multiplicação, sendo eles:

- **Step:** Para a matriz C , passando duas matrizes A e B como parâmetros, e os campos i, j da matriz C , a função calcula o valor final de $C_{i,j}$;
- **Multiply:** responsável por executar a multiplicação completa entre duas matrizes. Cada campo de C é passado para o método **Step**, invocado por meio de uma *goroutine* para permitir o paralelismo. Por fim, um *wait* é invocado no final da função para assegurar a completude das *goroutines*.

Dada a abstração de *slices*, não foi necessário nenhum uso de ponteiros nas funções responsáveis pela multiplicação, facilitando o acesso a trechos de um *array*. Assim, não é preciso uma gerência de memória explícita na hora de consumir e alterar esses *slices* já inicializados.

Código 6.2 – Código da função principal de multiplicação de matrizes em Go

```

package main

import (
    "time"
)

func main() {
    A, B, C := MatrixInt{}, MatrixInt{}, MatrixInt{}
    // inicializa as matrizes A, B com valores qualquer
    // inicializa todos os campos da matriz C com 0

    start := time.Now()
    C.Multiply(A, B)
    end := time.Now().Sub(start)
}

```

No código 6.2 da função principal, as três matrizes A , B e C são inicializadas. Para modificar a quantidade de threads utilizada pelo *runtime* do Go, foi utilizado a variável de ambiente `GOMAXPROCS`. Então, a chamada: `GOMAXPROCS=2 ./go_matrix 1000` executará a multiplicação de duas matrizes de tamanho 1000 por 1000 utilizando duas *threads* de sistemas operacionais.

6.3 COMPARAÇÃO COM OUTRA LINGUAGEM

Para realizar uma comparação de desempenho em relação ao tempo de execução, foi criado um código de multiplicação de matrizes em C++.

Código 6.3 – Código da função de multiplicação de matrizes em paralelo em C++

```

#include <cstdlib>

void MultiplyMatrix(int **A, int **B, int **C, int n, int m, int threads
, int id) {
    for (int i = id; i < n; i+=threads) {
        for (int k = 0; k < m; k++) {
            for (int j = 0; j < m; j++) {
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
            }
        }
    }
}

```

Inicialmente, a abordagem do código foi aproximar-se o máximo possível da implementação feita em Go, para que a diferença de desempenho não fosse influenciada pela

lógica da aplicação. Porém, dado a diferença de como funcionam fluxos concorrentes em cada linguagem, a abordagem inicial implementada em Go de utilizar uma *goroutine* para calcular cada elemento da matriz de saída não era viável em C++ devido ao custo de se criar um grande número de *threads* de sistema. Ainda assim, ambas soluções possuem a mesma complexidade de tempo, $O(N^3)$, visto que ambas utilizam o algoritmo clássico do problema, modificando apenas como é feito o paralelismo.

Existem diversas maneiras de particionar o trabalho para cada fluxo de execução. A utilizada foi de particionar a matriz *C* a partir da quantidade total de *threads*, e cada partição ser responsabilidade de uma determinada *thread*. Além dessa, ainda seria possível fazer uma linha da saída ser calculada por uma *thread* qualquer disponível. É possível observar no código 6.3 a implementação em que cada *thread* calcula um determinado número de linhas, com uma alternância entre as linhas a serem calculadas.

Código 6.4 – Código da função principal de multiplicação de matrizes em C++

```
#include <thread>

int main(int argc, char *argv[]) {
    int nthreads = 4; // n mero de threads
    int n = 100; // Matriz A: NxM
    int m = 100; // Matriz B: MxN
    int **A;
    int **B;
    int **C;
    // inicializa matrizes A e B com valores qualquer
    // inicializa todos os campos de C com valor 0

    // inicia contagem do tempo aqui
    auto ref_threads = new std::thread[nthreads];
    for (int i = 0; i < threads; i++) {
        ref_threads[i] = std::thread(MultiplyMatrix, A, B, C, n, m,
                                     nthreads, i);
    }
    for (int i = 0; i < threads; i++) {
        ref_threads[i].join();
    }
    // termina contagem do tempo aqui
}
```

O código 6.4 da função principal define dois valores: o tamanho da matriz e a quantidade de *threads* a ser utilizada. É realizado as inicializações das matrizes, e após isso é realizado a multiplicação, a partir da execução paralela em *threads* separadas. A função principal utiliza a função `join` para aguardar o encerramento de todas as *threads*.

6.4 OTIMIZAÇÃO DE ACESSO AO CACHE

Uma parte importante na hora de desenvolver algoritmos de alto desempenho, é avaliar corretamente a maneira que a transferência de memória entre as diversas camadas da hierarquia (desde a memória RAM até o cache do processador) está sendo realizada. No caso da multiplicação de matrizes temos um alto acesso a uma memória contínua do vetor.

Se observarmos a equação 6.1, temos que sempre precisamos acessar os índices $A_{i,k}$ e $B_{k,j}$. Porém, quando o último *loop* é a partir de k , temos diversos erros de acesso a memória, já que $B_{1,0}$ e $B_{2,0}$, por exemplo, não são dados contínuos na memória. Assim, aumenta-se a quantidade de transações entre os níveis de hierarquia da memória. Já se iterarmos em cima de j , temos que os valores serão acessados como $B_{1,0}$, $B_{1,1}$, seguindo o mesmo exemplo, e estes dados são contínuos.

6.5 MEDIÇÕES

O compilador utilizado para o C++ foi o `clang`, utilizando *flags* de otimização `Ofast`, semelhante a uma `O3` do `gcc`, com mais poucas otimizações. Em Go, foi utilizado o compilador padrão disponibilizado pelo Google. Todos os testes rodaram na mesma máquina, com sistema operacional Manjaro 18.0.4, com *kernel* do Linux versão 4.19.62-1-MANJARO para arquitetura `x86_64`. Foram realizados testes com os mesmos tamanhos de matrizes em ambas as linguagens, com tamanhos de 100 x 100, 1000 x 1000, 2000x2000, 3000x3000, 4000x4000 e 5000 x 5000. Utilizou-se 1, 2 e 4 núcleos de processamento simultâneos, na mesma máquina com CPU 3.8 GHz Intel Core i5-4670k (Haswell).

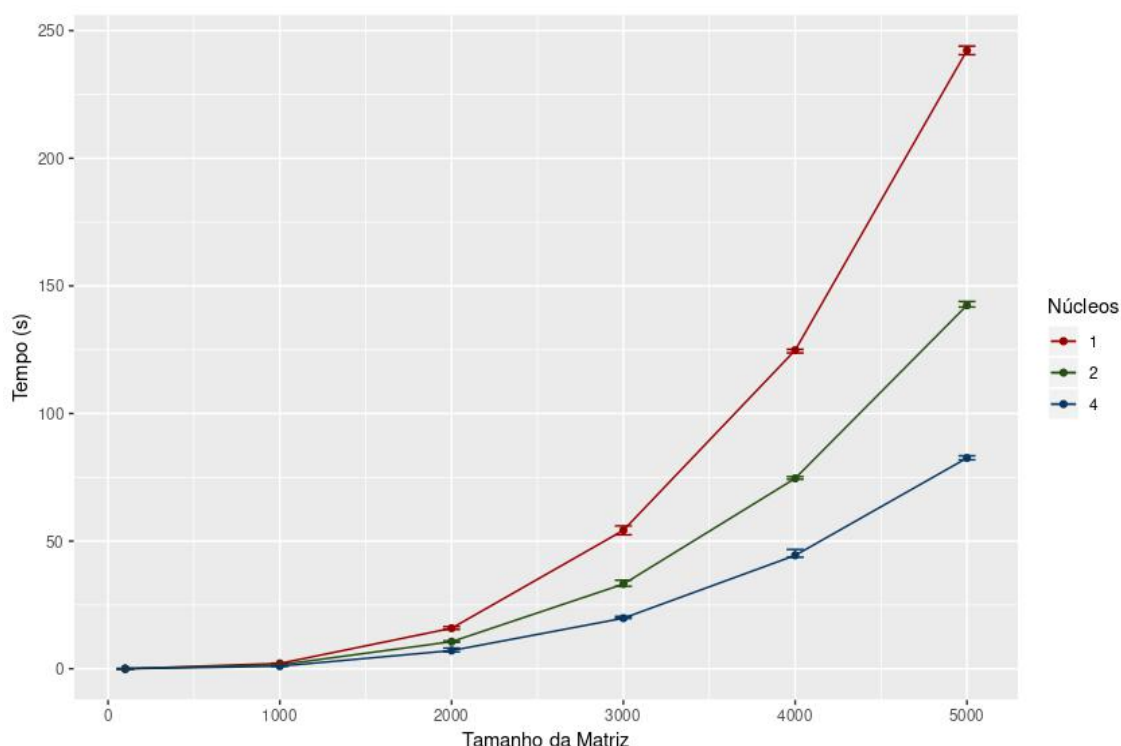
A fim de indicar a credibilidade das execuções, é possível ver nas tabelas o desvio padrão indicado pela coluna DP, e a porcentagem do coeficiente de variação a partir da coluna CV. Ambos os valores baixos indicam uma baixa variabilidade das execuções, o que indica uma boa precisão da amostra. Os valores obtidos podem ser observados na tabela 1. É possível ver para cada linguagem, o tempo médio de execução, o desvio padrão das amostras e suas covariâncias para todos os casos executados.

6.5.1 Desempenho por núcleo

É possível observar na figura 12 o tempo médio de execução a medida que se aumenta o tamanho das matrizes. No caso da matriz de maior tamanho, a média ficou próximo dos 7 minutos. O paralelismo do Go não foi capaz de atingir o seu máximo teórico, atingindo uma faixa de acelerar em 1.65x a medida que se dobra a quantidade de núcleos disponíveis. É possível ver o tempo médio de execução, o desvio padrão e a covariância na tabela 1.

Uma hipótese da baixa capacidade de paralelismo é o problema da escolha do algoritmo de bolsa de tarefas, em conjunto com os problemas de otimização de cache anteriormente ditos. Com o algoritmo de bolsa de tarefas, cada índice da coluna da matriz A pode

Figura 12 – Tempos de execução da solução em Go com otimização de acesso ao cache

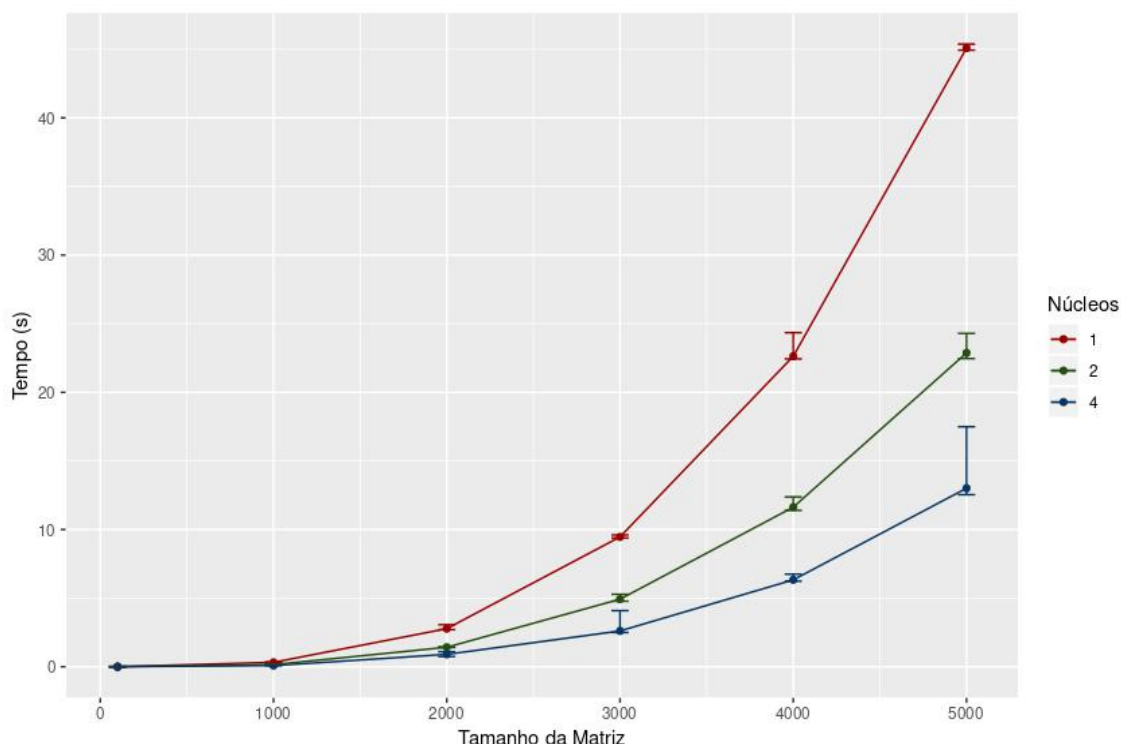


ser acessado por *threads* diferentes. Na implementação em C++, por exemplo, isso não ocorre, dado que os *loops* do j e do k não variam entre *threads*, apenas o *loop* da variável i , a mais externa do problema. Como há um aumento de erros de acerto ao cache na matriz A, que depende de qual *thread* pegará para processamento, a paralelização do problema pode acabar sendo comprometida por esse aumento de quantidade de transferência de memória necessária.

Na figura 13, é possível observar a soberania de desempenho da linguagem C++. Com um único núcleo, foi capaz de atingir uma velocidade de execução maior do que em quatro *threads* do Go. Apesar disso, é importante observar que o mérito não é exclusivo da linguagem. A implementação feita em C++ não observa o mesmo problema encontrado na implementação em Go, pois não há problemas de acesso ao cache em relação a matriz A. Além disso, como já visto anteriormente, grande parte da velocidade da linguagem se dá pelas otimizações feitas pelo compilador da linguagem. Há um espaço para um crescimento do desempenho da linguagem Go, se houver um foco nas otimizações tão poderosas como as existentes nas de C++.

Além do que foi observado, é importante notar que novamente neste caso, não foi atingido o paralelismo máximo teórico, apesar de suficientemente próximo, ficando por volta de 1.95x mais rápido o algoritmo a medida que a quantidade de núcleos disponíveis aumenta.

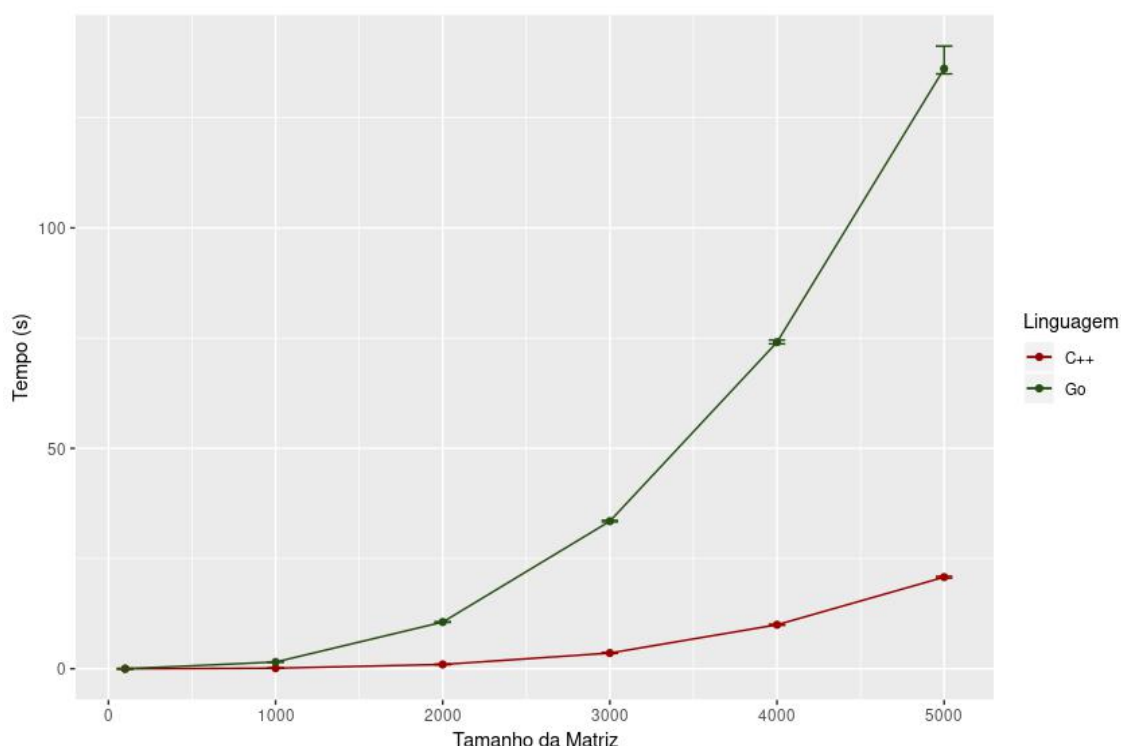
Figura 13 – Tempos de execução da solução em C++ com otimização de acesso ao cache



Por fim, temos na figura 14 as comparações de tempo de execuções das linguagens Go e C++ no cenário com quatro núcleos. É interessante ver que a velocidade de desempenho entre as duas soluções inicialmente são próximas, até eventualmente começarem a se afastar, devido tanto a perda de desempenho encontrada como ao fato de Go não ser tão otimizado. É possível também que, neste cenário, o custo de criação de múltiplas goroutines comece a ser caro, ultrapassando o valor de dez milhões de *goroutines* criadas. Ainda assim, o desempenho foi extremamente próximo do C++, e possível até de se melhorar, dado uma outra abordagem, como calcular cada linha inteira da matriz por cada *goroutine*.

Apesar do resultado não muito satisfatório comparado com a versão em C++, essa avaliação nos permitiu avaliar os resultados da linguagem perante as afirmativas de *goroutines* serem leves, podendo criar até milhões delas, e a ideia de que as abstrações oferecidas pela linguagem ainda permitirem códigos eficientes. E, como é possível ver na comparação da figura 14, o tempo de execução ficou muito próximo de C++ inicialmente, apesar de uma alta diferença depois. Temos que C++ é uma melhor escolha, em comparação com Go, em cenários de computação de alto desempenho. Ao mesmo tempo, Go se mostra uma alternativa viável para projetos que desejam ter um bom desempenho e poderosas abstrações de concorrência: nos cenários mais simples, a diferença do tempo de execução entre os dois programas é pouco, ficando ainda na casa dos segundos.

Figura 14 – Comparação das soluções em Go e C++, ambas com otimização de cache



O código de multiplicação de matrizes ainda é simples de compreender em C++, mas na medida que aumenta-se a complexidade dos algoritmos com a necessidade de uso de recursos de sincronismo, as responsabilidades para a execução correta do *software* sob o programador aumenta. Quanto maior o projeto, maior será sua complexidade, e explorar os recursos de concorrência, tidos como não triviais, é importantíssimo para permitir uma maior facilidade na manutenção de um projeto.

6.5.2 Considerações na compilação

O compilador utilizado para os códigos em C++ possui inúmeras otimizações de desempenho, algo que no compilador do Go não está presente. Em testes exploratórios, o desempenho do C++ sem utilizar o máximo das otimizações possuía um tempo de execução semelhante, podendo até ser maior, do que o tempo de execução do Go com seu compilador básico.

Assim, deve ser levado em questão as otimizações que o programa binário final possui. O tempo gasto em melhorias de desempenho dos compiladores de C++ é muito maior do que o do Go. Go é desenvolvido há onze anos, enquanto os compiladores de C e de C++ desde antes da década de 90.

Tabela 1 – Disposição da média, desvio padrão e covariância das amostras do tempo de execução da multiplicação de matrizes

Tamanho da Matriz		Quantidade de núcleos								
		1			2			4		
		Média	DP	CV	Média	DP	CV	Média	DP	CV
Go	100	0.0036	0.0007	20.7	0.0068	0.0012	18.4	0.0061	0.0014	23.1
	1000	2.65	0.0185	0.699	2.19	0.0202	0.923	1.53	0.00933	0.609
	2000	21.4	0.145	0.676	15.8	0.136	0.863	10.6	0.0380	0.359
	3000	73.7	0.487	0.660	53.8	0.381	0.709	33.5	0.0806	0.241
	4000	174	1.18	0.679	123	0.849	0.690	74.1	0.181	0.245
	5000	328	4.80	1.46	229	2.04	0.893	136	0.940	0.691
C++	100	0.0006	0.00008	13.6	0.0005	0.0001	22.4	0.0004	0.0001	30.4
	1000	0.467	0.00580	1.24	0.244	0.0044	1.81	0.125	0.0044	3.53
	2000	3.77	0.0282	0.748	1.92	0.0143	0.743	0.974	0.0107	1.09
	3000	14.5	0.119	0.821	7.10	0.0535	0.754	3.57	0.0409	1.15
	4000	37.4	0.354	0.947	19.1	0.150	0.783	9.98	0.0479	0.480
	5000	75.4	0.509	0.675	39.7	0.310	0.781	20.8	0.111	0.532

6.6 COMPARAÇÃO COM O MESMO ALGORITMO

Os resultados obtidos na multiplicação de matrizes em Go, utilizando uma espécie de “bolsa de tarefas”, usando as *goroutines* e o escalonador do Go, mostraram a inferioridade no desempenho de Go em relação ao C++. O paralelismo obtido foi muito longe do teórico. Para melhor analisar o resultado anterior obtido, foi realizada uma outra solução em Go que utiliza o mesmo algoritmo da implementação em C++. Assim, é possível comparar mais objetivamente as abstrações da linguagem.

Código 6.5 – Código da estrutura *Matrix* em Go com o mesmo algoritmo

```

package main

import (
    "sync"
)

type MatrixInt [][]int
func (C MatrixInt) Step(a, b MatrixInt, id, nthreads int, wg *sync.
    WaitGroup) {
    defer wg.Done()
    fmt.Println("starting", id)
    for i := id; i < len(a); i += nthreads {
        for k := 0; k < len(a[0]); k++ {
            for j := 0; j < len(a[0]); j++ {
                c[i][j] += a[i][k] * b[k][j]
            }
        }
    }
}

func (C MatrixInt) Multiply(a, b MatrixInt, threads int) {
    wg := &sync.WaitGroup{}
    for i := 0; i < threads; i++ {
        wg.Add(1)
        go c.Step(a, b, i, threads, wg)
    }
    wg.Wait()
}

```

A mudança relevante do código é realizada na estrutura de matriz. O código 6.5 modifica as funções `Step` e `Multiply`. A lógica de invocar a quantidade de *thread* fica na função `Multiply`, e a lógica do cálculo do trecho que uma *thread* calculará ficou na função `step`. Note que a separação por linhas foi a mesma escolhida no código 6.3, onde a *thread* inicia numa determinada linha, e a próxima calculada é baseada no número de *threads*.

Código 6.6 – Código da função principal em Go

```
package main

import (
    "time"
)

func main() {
    A, B, C := MatrixInt{}, MatrixInt{}, MatrixInt{}
    // inicializa o das matrizes A, B e C

    start := time.Now()
    C.Multiply(A, B, runtime.GOMAXPROCS(0))
    end := time.Now().Sub(start)
}
```

É possível ver que no código 6.6 como é feito a chamada da função de multiplicação. Para obter a quantidade total de *threads* de sistemas operacionais disponíveis, utiliza-se a função `GOMAXPROCS` da biblioteca `runtime` que é utilizada tanto para definir a quantidade de *kernel threads* quanto para pegar a quantidade atual. O primeiro argumento é a nova quantidade de *kernel threads* que o escalonador deve utilizar. Se o argumento for menor que 1, essa quantidade de *threads* não é alterada. A função sempre retorna o último valor lido antes da alteração.

A figura 15 mostra que o paralelismo obtido é mais próximo do valor teórico. O tempo de execução com uma única thread ficou semelhante ao executado na bolsa de tarefas, porém há uma melhoria no cenário de 2 *threads* e tamanho de matriz 5000: no algoritmo anterior atingiu-se quase 150 segundos, tendo uma média por volta de 145 segundos, enquanto neste algoritmo foi possível executar em média de 125 segundos.

Porém, ainda que fosse possível otimizar o paralelismo obtido, o problema base não se modificou: olhando ainda para o maior tamanho de matriz, a execução com um único núcleo ainda possui um tempo relativamente alto em comparação ao de C++. Então, apesar de resultados melhores que no algoritmo anterior, ainda encontram-se inferiores ao de C++. A figura 16 apresenta os resultados obtidos.

Figura 15 – Tempos de execução da solução em Go com otimização de cache e com o algoritmo de particionamento da matriz

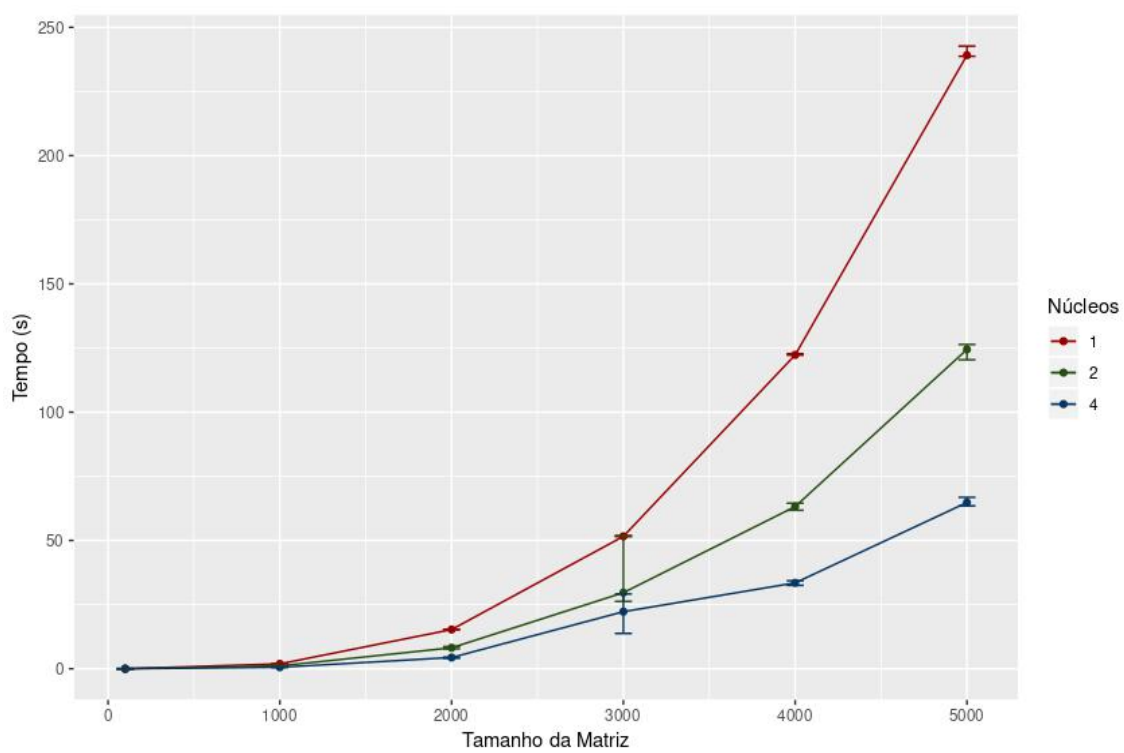
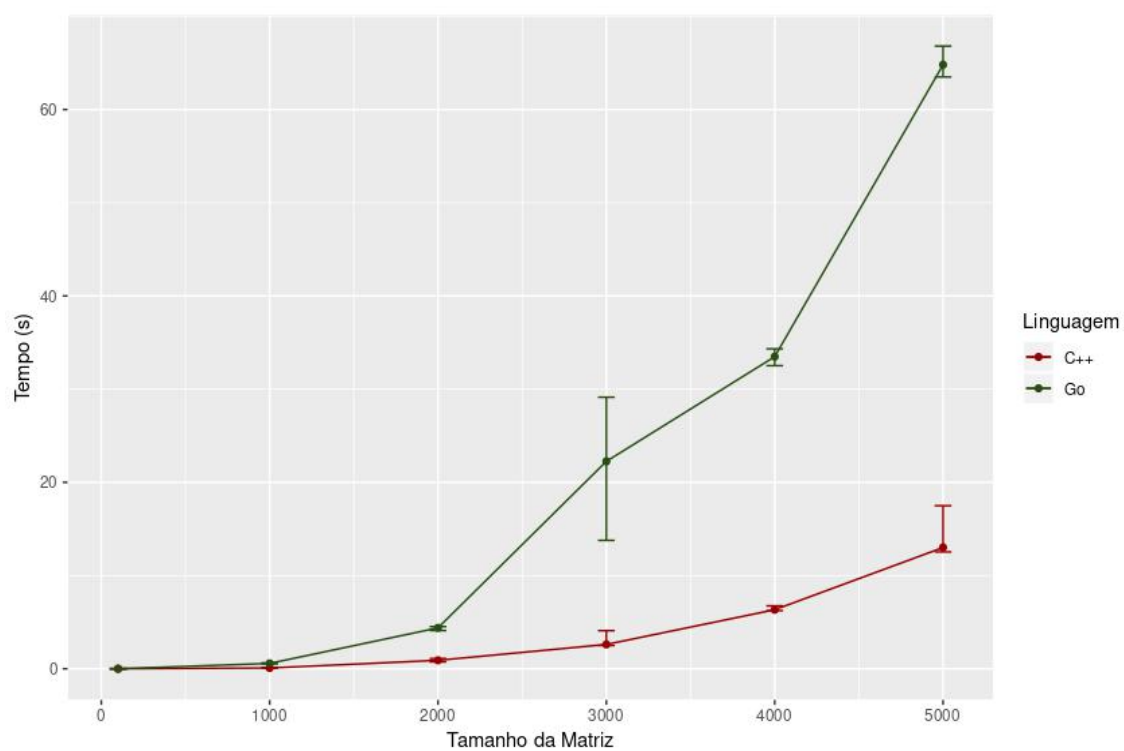


Figura 16 – Comparação das soluções em Go e C++, ambas com otimização de cache e com o mesmo algoritmo



7 CONCLUSÃO

O contexto atual e rumos das tecnologias modernas estão trazendo uma crescente necessidade em avanços na área de computação. A busca por melhor uso dos recursos e menor tempo de execução das aplicações, assim como a execução correta dessas aplicações, tornam-se cada vez mais relevantes. Em carros auto dirigidos, por exemplo, o uso de técnicas de inteligência artificial e visão computacional para fazer com que o carro consiga ter as melhores tomadas de decisões numa tarefa extremamente delicada pois envolve a vida de diversas pessoas, é combinado com uma vasta rede de informação compartilhada entre os carros. Um sistema distribuído extremamente complexo, com uma altíssima necessidade de desempenho, pois uma diferença de cálculo pode salvar vida. Outros exemplos são aplicações cada vez mais complexas como as de *streaming*, edição de texto, áudio e vídeo, e até mesmo poder jogar jogos remotamente, são grandes desafios que tangenciam ou estão completamente inseridos no mundo de computação concorrente.

A proposta deste trabalho foi explorar os mecanismos de concorrência da linguagem Go e suas aptidões para resolver problemas nessa velha área, cada vez mais exigida no mundo atual. Assim, foi possível analisar se uma linguagem teoricamente moderna, desenhada já com a intenção de atacar esses tipos de problemas, teria uma maneira de escrever código mais simples e compreensíveis ao mesmo tempo que ainda com um tempo de execução suficientemente alto.

Foi mostrado então os problemas que surgem de fluxos concorrentes sobre o ponto de vista de sincronizações: diferentes linhas de execução sendo processadas num ambiente em que diversos recursos são compartilhados, comumente uma determinada área de memória, exigem o uso adequado de mecanismos de sincronização, dado que várias situações levam à perda de corretude de um problema. No simples produtor consumidor, uma escrita descoordenada no buffer poderia levar a perda de uma informação importante, que jamais poderia ser compartilhada. Esse tipo de resultado pode ser catastrófico, desde uma perda comercial para uma empresa indo até o nível de colocar vidas em riscos. Os modelos de concorrência devem ser mais seguros e simples de utilizar, e Go não somente propõe como de fato disponibiliza ferramentas intuitivas e menos dispostas a erros.

Com as implementações dos problemas selecionados, foi possível se observar como as diversas funcionalidades de concorrência da linguagem se complementam, permitindo uma implementação mais simplificada. A ideia é se aproximar mais de um modelo de troca de mensagens, que permita comunicar o estado atual das diversas entidades do programa e se afastar da ideia de observar um mesmo local de memória por diversos fluxos para entender qual o estado atual do problema. Na solução proposta para o problema do barbeiro dorminhoco, os clientes e barbeiro se comunicam por meio de canais para indicar tanto o estado atual (se há cliente pronto, ou se o barbeiro já despertou), como também

para indicar a quantidade atual de assentos. Com o ordenamento que canais provê, o problema ainda recebe uma fila que não precisou ser gerenciada, nem mesmo criada: a própria primitiva de canal já fornece esse mecanismo.

Do ponto de vista de desempenho, Go apresentou tempos de execução mais lentos que numa execução semelhante em C++. Tanto nas execuções paralelas, com mais de uma *thread*, quanto nas execuções com apenas uma *thread*, como é possível observar na tabela de tempo de execuções do capítulo 4. Apesar de mais lento que C++, Go consegue se aproximar do uso máximo de recursos, tendo um aumento quase linear de desempenho a medida que se aumenta a oferta de processadores disponíveis para execução.

Nas aplicações distribuídas, com os mecanismos de concorrência oferecidos torna-se simples a escrita e compreensão de códigos destes tipos de aplicações. O modelo de troca de mensagens entre os sistemas é mais facilmente visualizado no próprio código a partir dos canais. A biblioteca básica provê também ferramentas prontas para se utilizar até mesmo em sistemas finais de empresas. Em comparação com outras linguagens, é normalmente trabalho da comunidade o ferramental para o protocolo HTTP. Python, Javascript, C, C++, Java, Lua, entre diversas outras dependem de ferramentas não integradas a linguagem para escritas de aplicações web. *Sockets*, resolução de DNS, conexões TCP/IP, tudo isso é disponibilizado por Go, oferecendo uma grande facilidade tanto no desenvolvimento de aplicações finais, quanto de bibliotecas que são facilmente integradas dado que todas utilizam a mesma base.

Para a solução de corrotinas, nota-se a ausência de recursos como *generics*. Para trazer uma generalização máxima do problema, é necessário diversos tratamentos direto das variáveis junto com a lógica do algoritmo a ser resolvido com corrotina. Com a proposta de manter-se simples e direta, abre-se mão de ferramentas que podem ser úteis para criar complementos para a linguagem.

Go mostra-se cumprir a proposta de criar uma linguagem concorrente. Os tipos estáticos, as ferramentas síncronas de troca de mensagem (podendo ser assíncronas caso necessário), as diversas bibliotecas básicas para diferentes tipos de problema, todas essas abstrações oferecem uma segurança maior na escrita do código, sem abdicar do desempenho. Com a linguagem caminhando para sua segunda versão, é importante avaliar quais serão as simplificações a serem abdicadas e quais serão mantidas, com o preço talvez do aumento da complexidade tanto do código escrito quanto do atual compilador do Go.

Go, apesar de mais lento, pode-se argumentar que é uma linguagem mais simples de se desenvolver. Os seus recursos de concorrência permitem a criação de código eficientes, ainda que não com desempenho máximos. O tratamento de erro, manipulação dos elementos de concorrência, funções de primeira ordem, interfaces e métodos, baixo tempo de compilação, entre os outros recursos da linguagem entregam uma facilidade maior para o desenvolvimento de aplicações complexas em sistemas distribuídos. Não é em toda situação que existe a necessidade do máximo de desempenho: muitos programas de linha de

interface, APIs (*application programming interface*) para outros sistemas, programas com interface gráfica, não requisitam de toda o desempenho dado por linguagens como C++ e seus compiladores otimizados. E, mesmo que Go não atinja o desempenho completo de outras linguagens, ainda é suficientemente eficiente: seu desempenho se aproxima de C++ no cenário sem otimizações numa tarefa que Go não se propõe ser o mais eficiente, que é de tarefas com alto grau de paralelismo. Assim, Go torna-se uma alternativa viável para cenário onde não se pode abdicar de desempenho, ao mesmo tempo que deseja-se construir códigos de mais fácil entendimento e manutenabilidade do que linguagens mais eficientes.

REFERÊNCIAS

BALLHAUSEN, H. Fair solution to the reader-writer-problem with semaphores only. **arXiv preprint cs/0303005**, 2003.

BINET, S. Go-HEP: writing concurrent software with ease and go. **Journal of Physics: Conference Series**, IOP Publishing, v. 1085, p. 052012, sep 2018. Disponível em: <https://doi.org/10.1088%2F1742-6596%2F1085%2F5%2F052012>.

BRYANT, R. E.; RICHARD, O. D.; RICHARD, O. D. **Computer systems: a programmer's perspective**. [S.l.]: Prentice Hall Upper Saddle River, 2003. v. 281.

CORPORATION, O. **JDK 1.1 for Solaris Developer's Guide**. [S.l.]: Sun Microsystems, 2000.

CUNHA, A. G. D. **Dicionário etimológico da língua portuguesa**. [S.l.]: Lexikon Editora, 2019.

DIJKSTRA, E. W. Information streams sharing a finite buffer. In: **Inf. Proc. Letters**. [S.l.: s.n.], 1972. v. 1, p. 179–180.

DONOVAN, A. A.; KERNIGHAN, B. W. **The Go programming language**. [S.l.]: Addison-Wesley Professional, 2015.

GUIO, J. **Códigos do Trabalho de Conclusão de Curso**. 2019. Disponível em: <https://github.com/jonnyguio/tcc>.

ISO. **ISO/IEC 14882:2011 Information technology — Programming languages — C++**. Terceira. International Organization for Standardization, 2011. Disponível em: <https://www.iso.org/standard/50372.html>.

JOSHI, K. **The Scheduler Saga**. 2018. Disponível em: <https://www.youtube.com/watch?v=YHRO5WQGh0k>. Acesso em: 12 jul.2019.

KERRISK, M. **pthread_create(3)LinuxProgrammer'sManual**.2018.Disponvelem : <>. Acesso em: 12 jul.2019.

LAMPORT, L. Time, clocks and the ordering of events in a distributed system. **Communications of the ACM** 21, 7 (July 1978), 558-565. Reprinted in several collections, including **Distributed Computing: Concepts and Implementations**, McEntire et al., ed. IEEE Press, 1984., p. 558–565, July 1978. 2000 PODC Influential Paper Award (later renamed the Edsger W. Dijkstra Prize in Distributed Computing). Also awarded an ACM SIGOPS Hall of Fame Award in 2007. Disponível em: <https://www.microsoft.com/en-us/research/publication/time-clocks-ordering-events-distributed-system/>.

MARLIN, C. D. **Coroutines: a programming methodology, a language design and an implementation**. [S.l.]: Springer Science & Business Media, 1980.

MOURA, A. L. D.; IERUSALIMSKY, R. Revisiting coroutines. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, v. 31, n. 2, p. 6, 2009.

PADUA, D. **Encyclopedia of parallel computing**. [S.l.]: Springer Science & Business Media, 2011.

PIKE, R. et al. **The Go Programming Language Specification**. Disponível em: <<https://golang.org/ref/spec>>. Acesso em: 10 jun.2019.

PIKE, R. et al. **Perguntas frequentes sobre Go**. 2019. Disponível em: <<https://golang.org/doc/faq>>. Acesso em: 01 jul.2019.

PIKE, R. et al. **Página principal do Go**. 2019. Disponível em: <<https://golang.org/>>. Acesso em: 01 jul.2019.

PRANSKEVICHUS, E.; SELIVANOV, Y. **What's New In Python 3.5**. 2015. Disponível em: <<https://docs.python.org/3.5/whatsnew/3.5.html>>. Acesso em: 15 jul.2019.

STEVENS, W. R. **UNIX network programming: Interprocess Communication**. [S.l.]: Prentice Hall Ptr, 2009.

APÊNDICES

APÊNDICE A – CÓDIGOS DE EXEMPLOS DA LINGUAGEM GO

Código A.1 – Exemplo de tipos booleanos

```
package main

import "fmt"

func main() {
    a := true
    fmt.Println("Valor de a:", a)
}
```

Código A.2 – Exemplo de tipos numéricos

```
package main

import "fmt"

func main() {
    var a int64
    var b float64
    c := 10
    a = 20
    b = 10.999999999999
    fmt.Println(int(a) + c)
    fmt.Println(float64(a) + b)
}
```

Código A.3 – Exemplo de texto

```
package main

import "fmt"

func main() {
    a := "Hello"
    b := "          !\n"
    fmt.Printf("%s", a + b)
}
```

Código A.4 – Exemplo de array e slice

```
package main

import "fmt"

func main() {
    odds := [5]int{1, 3, 5, 7, 9}

    var firstThreeOdds []int = odds[0:3]
    fmt.Println(firstThreeOdds)
}
```

Código A.5 – Exemplo de estrutura

```
package main

import "fmt"

type foo struct {
    Bar string
    Baz int
}

func multiplication(a int, x int) int {
    return a * x
}

func main() {
    v := &foo{"Hello", 1}

    fmt.Printf("%+v\n%d\n", v, multiplication(v.Baz, 10))
}
```

Código A.6 – Exemplo de interface e método

```
package main

import "fmt"

type Vec interface {
    Sum(Vec) Vec
    Minus(Vec) Vec
}

type Vector2 struct {
    x, y float64
}

func (v Vector2) Sum(v2 Vector2) Vector2 {
    return Vector2{v.x + v2.x, v.y + v2.y}
}

func (v Vector2) Minus(v2 Vector2) Vector2 {
    return Vector2{v.x - v2.x, v.y - v2.y}
}

func main() {
    var v1 Vec
    v1 = Vector2{1, 1}
    v2 := Vector2{10, 10}
    fmt.Println(v1.Sum(v2))
}
```

Código A.7 – Exemplos de mapas

```
package main

import "fmt"

func main() {
    urls := map[string]string{
        "http": "http://example.com",
        "https": "https://example.com",
    }
    fmt.Println(urls["http"])
}
```

Código A.8 – Diferentes usos das estruturas condicionais clássicas

```
package main

import "fmt"

func main() {
    a := 10
    if a < 10 {
        fmt.Println("'a' menor que 10")
    }
    if a < 5 {
        fmt.Println("'a' menor que 5")
    } else {
        fmt.Println("'a' maior que 5")
    }
    if b := a / 2; b < 10 {
        fmt.Println("'a/2' menor que 10")
    }
}
```

Código A.9 – Exemplo do uso de for

```
package main

import "fmt"

func main() {
    a := [5]int{1, 2, 3, 4, 5}
    for i := 0; i < len(a); i++ {
        fmt.Printf("a[%d]: %d\n", i, a[i])
    }
}
```

Código A.10 – Iterando em um array por meio da palavra reservada `range`

```

package main

import "fmt"

func main() {
    a := [5]int{1, 2, 3, 4, 5}
    for i, v := range a {
        fmt.Printf("a[%d]: %d\n", i, v)
    }
}

```

Código A.11 – Exemplo de interface para comunicação remota

```

package main

type Connection interface {
    Read([]byte) (int, error)
    Write([]byte) error
}

type TCP struct{}

func SendStringToConn(value string, conn Connection) {
    err := conn.Write([]byte(value))
    if err != nil {
        panic(err)
    }
}

func (t TCP) Read(dest []byte) (int, error) {
    // implement read from tcp socket
    return 0, nil
}

func (t TCP) Write(input []byte) error {
    // implement write to tcp socket
    return nil
}

func main() {
    tcpSocket := TCP{}
    a := "Hello world!"
    SendStringToConn(a, tcpSocket)
}

```